MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. T.R. No. 874 November 1985

BUILD:

A Tool For Maintaining Consistency In Modular Systems

by

Richard Elliot Robbins

Abstract

BUILD is a tool for keeping modular systems in a consistent state by managing the construction tasks (e.g. compilation, linking etc.) associated with such systems. It employs a user supplied system model and a procedural description of a task to be performed in order to perform the task. This differs from existing tools which do not explicitly separate knowledge about systems from knowledge about how systems are manipulated.

BUILD provides a static framework for modeling systems and handling construction requests that makes use of programming environment specific definitions. By altering the set of definitions, BUILD can be extended to work with new programming environments and to perform new tasks.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grant MCS-8117633, in part by the International Business Machines Corporation, and in part by Honeywell Information Systems, Incorporated.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, either expressed or implied, of the Department of Defense, of the National Science Foundation, of the International Business Machines Corporation, or of Honeywell Information Systems, Incorporated.

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on June 3, 1985 in partial fulfillment of the requirements for the degree of Master of Science.

Dedicated To

The memory of my grandmothers, Ruth and Esther.

Acknowledgments

I would like to acknowledge the role that my thesis advisor, Dick Waters, played in the work that this report is based on: the importance of his guidance and encouragement cannot be over-estimated. I would also like to thank Bob Zieve, Jacques Bouvard and Honeywell Information Systems for supporting this research. Finally, I would like to thank Donna Gurshel, Dave Wheeler, Pete Sterpe, Suzanne Witty, Marc Zissman, Dave Kravitz, Sam Levitin, and all of the people who were even remotely connected with the Honeywell Day Care Center for providing the friendship and moral support that allowed me to see this project to its completion.

Table of Contents

List of Figures

1. Introduction

Many programming languages encourage the development of modular systems by allowing the independent compilation of modules (ADA [Ada 83], C [Kernighan and Ritchie 78], CLU [Liskov 81], Common-Lisp [Steele 84], Mesa [Mitchell 79]). This feature can be exploited to minimize the amount of compilation that needs to be done when some part of a system is changed. However, as systems become larger it becomes difficult to know exactly which modules need to be recompiled when one changes. It is important that the correct modules be recompiled and relinked -- a bug caused by ignoring a module that should be rebuilt can be very difficult to find. This problem is called the consistent construction problem.

This report describes BUILD, a tool that reconstructs system modules in order to ensure that they are kept in a consistent state. BUILD does not modify source modules and will not rid systems of problems that require source code revision. However, BUILD can handle the many instances where some portion of a system needs to be recompiled, relinked, or somehow reprocessed in order to eliminate inconsistency.

There are many tools that manipulate systems by reconstructing inconsistent parts. Chapter 2 presents MAKE [Feldman 79] and DEFSYSTEM [Weinreb and Moon 81], two representative tools, and discusses some of their weaknesses. The fundamental problem with MAKE, DEFSYSTEM, and all similar construction directive based tools is that they operate on systems by using user supplied lists of construction directives. These lists are difficult to understand. BUILD provides the same functionality as existing tools but does so without requiring users to list construction steps.

BUILD derives the construction steps needed to produce a module from user supplied *system models*. These models specify how modules reference each other instead of how they are constructed. BUILD uses the reference information to determine how modules depend on each other and how a change to one module will effect another. For instance, if a system model specifies that $module_1$ refers to macros defined in $module_2$, then BUILD can infer that a change to $module_2$ implies that $module_1$ should be recompiled. Chapter 3 discusses system models and chapters 4, and 5 explain how BUILD uses system models to perform construction.

The major strength of BUILD's reference based modeling system over a construction directive based system is that it provides a higher level language for describing system structure. Because it eliminates low level construction detail and allows explicit declaration of high level system relationships, a reference based model is easier to understand and provides more information than its construction directive based counterpart.

BUILD separates knowledge about systems from knowledge about how systems are manipulated. The term *task* is used to refer to a construction process such as compilation or linking that BUILD may be called upon to perform. BUILD uses *task descriptions* to specify how to perform construction tasks and how the various kinds of references that appear in system models may effect the construction required to perform the task. Using the example from the previous paragraph, BUILD's task description for compilation allows it to realize that while a change to $module_2$ implies that $module_1$ should be recompiled, a change to $module_1$ does not imply that $module_2$ should be recompiled.

TINYCOMP

TINYCOMP is an example of a modular system, it will be used throughout this report to present different aspects of system construction tools (this example was adapted from one used by Feldman [Feldman 79]). TINYCOMP has two major modules, a parser and a code generator. The parser is built by YACC, a parser generating tool [Johnson 78a]. The code generator is implemented in C [Kernighan and Ritchie 78]. The parser and code generator use a common set of definitions for shared data structures. These definitions are combined with the source programs during compilation. The compiled programs are linked with a library that is also subject to change. Figure 1-1 depicts TINYCOMP's inter-module reference pattern and figure 1-2 depicts TINYCOMP's construction process.

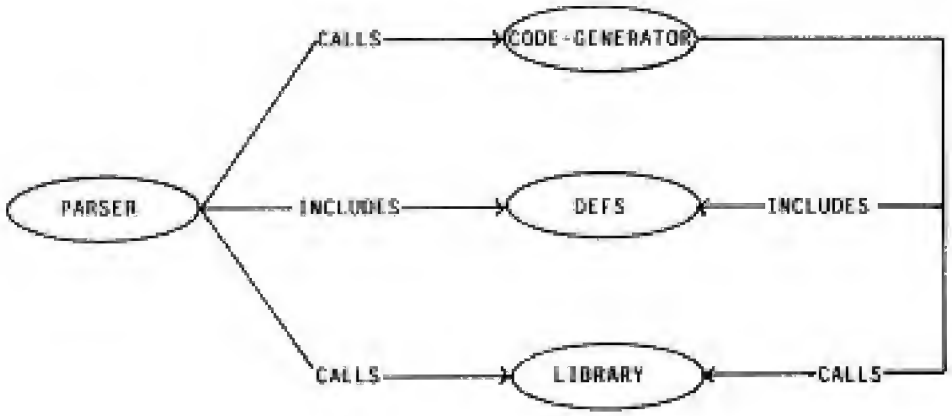

Figure 1-1: TINYCOMP Inter-Module Reference Graph

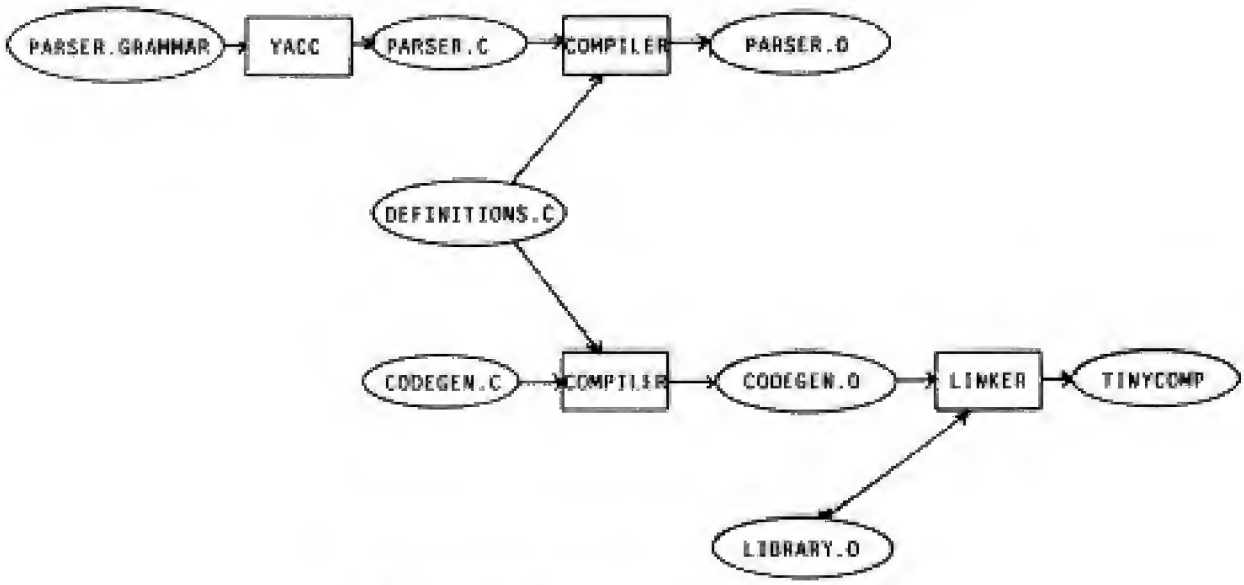

Figure 1-2: Construction Graph For TINYCOMP

Reference Based System Models

Compare figure 1-3 which contains the MAKE directives for TINYCOMP, and figure 1-4 which contains the BUILD system model for TINYCOMP. While the MAKE directives encode TINYCOMP's construction graph, BUILD's system model encodes TINYCOMP's reference graph.

A reference model can be used in place of a construction directive list because all of the information about construction present in such a list can be derived from a reference model. Consider the third MAKE directive for TINYCOMP:

```
CODEGEN.O: CODEGEN.C DEFINITIONS.C
        CC -C CODEGEN.C          # -C COMPILES
```

This expresses that `CODEGEN.O` is produced by compiling `CODEGEN.C`, and that if either `CODEGEN.C` or `DEFINITIONS.C` changes, then `CODEGEN.C` needs to be recompiled. This construction dependency exists because `CODEGEN.C` is combined with `DEFINITIONS.C` when it is compiled to produce `CODEGEN.O`.

In contrast, the reference based model specifies that `CODE-GENERATOR` *includes* `DEFS`:

```
(:INCLUDES CODE-GENERATOR DEFS)
```

BUILD's description for compilation contains the knowledge that the `:INCLUDES` reference implies a compilation construction dependency between *including* and *included* files.

```
PARSER.C: PARSER.GRAMMAR
        YACC PARSER.GRAMMAR      #YACC MAKES Y.TAB.C
        MV Y.TAB.C PARSER.C      #RENAME Y.TAB.C

PARSER.O: PARSER.C DEFINITIONS.C
        CC -C PARSER.C           # -C COMPILES

CODEGEN.O: CODEGEN.C DEFINITIONS.C
        CC -C CODEGEN.C          # -C COMPILES

TINYCOMP: CODEGEN.O PARSER.O LIBRARY.O
        CC CODEGEN.O PARSER.O LIBRARY.O -O TINYCOMP # -O LINKS
```

Figure 1-3: MakeFile For TINYCOMP

```
(DEFMODEL TINYCOMP
  (:MODULE DEFS :C-SOURCE "DEFINITIONS")
  (:MODULE PARSER :YACC-GRAMMAR "PARSER")
  (:MODULE CODE-GENERATOR :C-SOURCE "CODEGEN")
  (:MODULE LIBRARY :C-OBJECT "LIBRARY")

  (:INCLUDES PARSER DEFS)
  (:INCLUDES CODE-GENERATOR DEFS)
  (:CALLS PARSER LIBRARY)
  (:CALLS PARSER CODE-GENERATOR)
  (:CALLS CODE-GENERATOR LIBRARY))
```

Figure 1-4: BUILD Model For TINYCOMP

Task Descriptions

Upon receipt of a request to perform a task, BUILD derives a task graph which models the construction steps and dependencies necessary to perform the task. (Chapter 4 presents BUILD task models and chapter 5 explains how task models are derived from system models.) Once the task model has been derived, BUILD analyzes it in order to determine which components have changed and what steps are needed in order to satisfy the task request.

BUILD provides a static framework for modeling systems and handling construction requests that makes use of programming environment specific definitions. New tasks can be added to BUILD's repertoire by altering the set of definitions.

For example, figure 1-5 contains the forms needed to define a task called :LIST-SOURCE-CODE which produces formatted listings of the source modules of a Lisp system. (This example will be explained in detail in chapter 5.) The first form allows BUILD to represent the processing needed to list a single Lisp source file. The second form tells BUILD what to do when a :LIST-SOURCE-CODE request is received. The last two forms tell BUILD about the implications of the references :CALLS and :MACRO-CALLS upon the :LIST-SOURCE-CODE task.

Since task definitions are separate from system models, new tasks can be performed on existing models without additional effort. For instance, once :LIST-SOURCE-CODE has been defined, BUILD will be able to handle requests to format the source code for existing systems without changing any system models. Construction directive based tools cannot be extended in a similar manner.

```
(DEFINE-PROCESS-TYPE :LIST-LISP-SOURCE
    ((SOURCE :LISP-SOURCE :SINGLE))
    ((LISTING :PRESS :SINGLE SOURCE))
    OUTPUT-STREAM
    (FORMAT OUTPUT-STREAM "~%LIST ~A"
            (PATHNAME-MINUS-VERSION SOURCE))
  (FORMAT OUTPUT-STREAM "~%LISTING ~A" SOURCE)
  (LIST-LISP-FILE SOURCE LISTING))

(DEFINE-REQUEST-HANDLER (:LIST-SOURCE-CODE :LISP-SOURCE :PRE)
                        (SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE :LIST-LISP-SOURCE) LISTING)))

(DEFINE-REFERENCE-HANDLER ((:MACRO-CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:LIST-SOURCE-CODE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LIST-SOURCE-CODE CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:LIST-SOURCE-CODE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LIST-SOURCE-CODE CALLED-NODE))
```

Figure 1-5: Definition For :LIST-SOURCE-CODE

2. System Construction Tools

This chapter focuses on two tools that were designed to aid in the management of the consistent construction problem. Before they are presented some terminology that will be used throughout this report is introduced.

Different programming environments are geared to operate upon different kinds of objects. For instance, some environments are designed to operate on files, and others on functions. The term *grain* will be used to refer to the objects manipulated in a programming environment -- regardless of their nature.

The terminology introduced in this paragraph will be used to refer to the kinds of grains that are manipulated during the construction process. *Source* grains are the components that are produced by people and not programs (e.g., programming language source code). Source grains are manipulated by programs to produce *derived* grains (e.g., object code). Grains that are the final products of the construction process are called *goal* grains (e.g., executable images of programs). While goal grains are usually derived grains, they can also be source grains. Derived grains that are not goal grains are called *intermediate* grains (e.g., object code that requires linking in order to form executable images).

2.1 MAKE

MAKE [Feldman 79], available as part of UNIX[1], is a simple tool for managing systems that has received widespread use. MAKE is driven by sets of construction directives that form "recipes" for constructing systems. These directives are stored in a text file called a MakeFile and have the form:

```
TARGET-GRAIN : INGREDIENT-GRAIN-1 INGREDIENT-GRAIN-2 ...
        COMMAND-1
        COMMAND-2
          :
          ;
```

Each entry declares that *TARGET-GRAIN* depends on each of the grains to the right of the colon. The command sequence below the construction dependency declaration line is executed in order to construct *TARGET-GRAIN*. There are no constraints placed on the commands which can appear in the command sequence. Furthermore, there are no ordering rules for MakeFile entries.

MAKE has a simple macro substitution facility. A macro is defined in the following manner:

```
MACRO-NAME=MACRO-EXPANSION
```

Any instance of *MACRO-NAME* enclosed within parentheses and preceded by a dollar sign (i.e., $(*MACRO-NAME*)) is replaced by the text *MACRO-EXPANSION* when the MakeFile that includes the macro definition is processed. The definition for a macro must precede all of its uses.

A Small Example -- TINYCOMP

Figure 2-1 depicts the construction process for TINYCOMP and figure 2-2 contains a corresponding MakeFile. Given the MakeFile, MAKE will perform the appropriate construction when TINYCOMP components change. For instance, a change to PARSER.GRAMMAR will cause a new parser to be derived, compiled, and linked. A change to CODEGEN.C will cause CODEGEN.C to be compiled and linked. A change to DEFINITIONS.C will cause PARSER.C and CODEGEN.C to be compiled and linked. Finally, a change to LIBRARY.O will cause linking but no compiling.

---

[1] UNIX is a trademark of Bell Laboratories

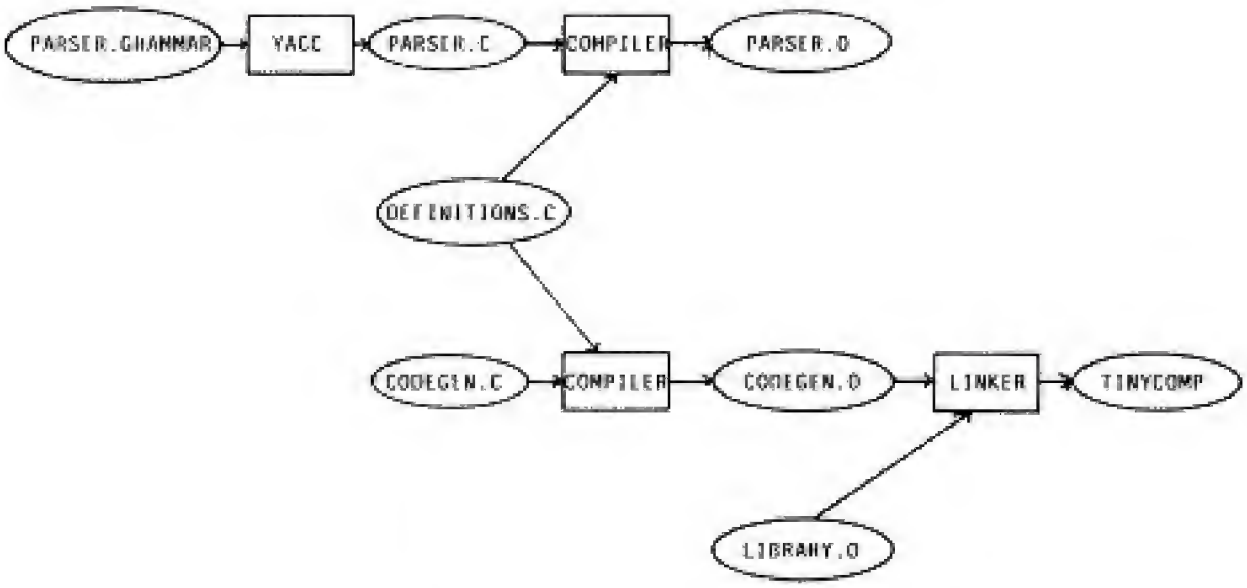

Figure 2-1: Construction Graph For TINYCOMP

```
PARSER.C: PARSER.GRAMMAR
        YACC PARSER.GRAMMAR      #YACC MAKES Y.TAB.C
        MV Y.TAB.C PARSER.C      #RENAME Y.TAB.C

PARSER.O: PARSER.C DEFINITIONS.C
        CC -C PARSER.C           # -C COMPILES

CODEGEN.O: CODEGEN.C DEFINITIONS.C
        CC -C CODEGEN.C          # -C COMPILES

TINYCOMP: CODEGEN.O PARSER.O LIBRARY.O
        CC CODEGEN.O PARSER.O LIBRARY.O -O TINYCOMP # -O LINKS
```

Figure 2-2: MakeFile For TINYCOMP

The MakeFile entries are interpreted in the following manner:

`PARSER.C: PARSER.GRAMMAR ...`
PARSER.C depends on PARSER.GRAMMAR. It is created by running YACC on PARSER.GRAMMAR.

`PARSER.O: PARSER.C DEFINITIONS.C ...`
PARSER.O depends on PARSER.C and DEFINITIONS.C. It is created by recompiling PARSER.C.

`CODEGEN.O: CODEGEN.C DEFINITIONS.C ...`
CODEGEN.O depends on CODEGEN.C and DEFINITIONS.C. It is created by recompiling CODEGEN.C.

`TINYCOMP: CODEGEN.O PARSER.O LIBRARY.O ...`
TINYCOMP depends on CODEGEN.O, PARSER.O, and LIBRARY.O. It is created by relinking the system.

The Construction Process

MAKE is invoked with the following UNIX command line template (brackets indicate optional fields):

```
MAKE [-f MAKEFILE] [OPTION ... ] [TARGET-GRAIN]
```

MAKEFILE Specifies the name of the file containing the construction directives. if no **-f** option is used then MAKE uses the file named **MAKEFILE** in the working directory.

OPTION Specifies options like *print but do not execute the command sequences* or *update the modified date of the targets without executing any command sequences.*

TARGET-GRAIN Specifies the name of the target grain to be processed. If *TARGET-GRAIN* is not specified then MAKE will process the first target grain named in the MakeFile.

MAKE begins by constructing a dependency graph from the selected MakeFile. Each node in the graph corresponds to a grain mentioned in the MakeFile. The children of a node represent the grains that the grain represented by the node depends on. A request to *make* a target grain is processed by doing a depth-first walk of the graph starting with the node that corresponds to the target. At each node visited, any grains that are missing or whose children have changed are updated.

MAKE compares the creation dates of a target grain and its ingredient grains as an approximate means of noting when changes occur. For instance if **TARGET-1** depends on **INGREDIENT-1** then MAKE will assume that **INGREDIENT-1** has changed if and only if its creation date is after the creation date of **TARGET-1**. Since UNIX allows file creation dates to be modified by users, it is possible to fool MAKE by changing file attributes. However, since most people do not change file attributes, the MAKE mechanism is reasonable.

Without information about how an ingredient has changed, MAKE cannot determine whether a change is significant or not. Therefore, MAKE pessimistically assumes that every change to an ingredient grain will effect the target grain, and it will always reconstruct a target when one of its ingredients has changed. Figure 2-3 contains the MAKE construction algorithm written in Lisp.

```
(DEFUN MAKE (NODE)
  (DOLIST (CHILD (GET-CHILDREN NODE))
    (MAKE CHILD))
  (IF (OR (NON-EXISTENT-P NODE) (CHILDREN-CHANGED-P NODE))
      (UPDATE NODE)))

(DEFUN CHILDREN-CHANGED-P (NODE)
  (< (CREATION-DATE NODE)
     (APPLY #'MAX
            (MAPCAR #'GET-CREATION-DATE (GET-CHILDREN NODE)))))
```

Figure 2-3: MAKE Construction Algorithm

An Extended Example -- LINT

The LINT system [Johnson 78b] is presented as an extended example of using MAKE. LINT examines C source programs and detects bugs that most C compilers cannot. It is also sensitive to constructs that are legal but may not be portable.

LINT consists of a UNIX shell script driver, a set of LINT Library files, and two C programs. Before programs are processed by the first C program (i.e., the first pass of LINT), they are processed by the C pre-processor, which handles macro expansion and some compiler directives.

After being processed by the C pre-processor, programs are sent to the first pass of LINT. This pass does lexical analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. An intermediate file that consists of lines of ASCII text is produced. Each line contains an external identifier name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees. Comments about local problems are produced as detected. The information about external names is collected in the intermediate file.

LINT libraries are collections of definitions of external names that are appended to the intermediate file generated by the first pass of LINT. They are used to provide LINT with a set of definitions for commonly used external names without processing the source that contains the definitions. The most commonly used libraries contain the definitions for the functions that are supplied by the UNIX C run time environment. Users can create their own libraries of commonly used names in order to alleviate repeated processing.

After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second pass of LINT then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

Figure 2-4 contains the MakeFile for LINT. The primary point of this example is that MakeFiles, even for medium sized systems like LINT, are difficult to understand. The BUILD description mechanism introduced in chapter 3 provides a much simpler way to describe systems.

The first part of the LINT MakeFile contains macro definitions. These definitions are used to specify directories (e.g., `M`), compilation flags (e.g., `CFLAGS`), and to group files (e.g., `LINTLIBS`). The target `ALL` is used to name the major subsystems of the LINT. The next cluster of specifications manages the first pass of LINT. There is an entry for each library file provided with LINT. Each of these specifies that a LINT library file is dependent upon a library source file and the first pass of LINT. Libraries depend on the first pass of LINT because they are constructed by it. The targets that specify management for the second pass of LINT are `LPASS2` and `LPASS2.O`.

The `LINTALL`, `INSTALL`, `SHRINK`, and `CLEAN` targets are not grains at all, rather, they are used to initiate installation and removal of LINT. A request to *make* any of these will always result in the associated command sequence being executed because the corresponding files do not exist in the UNIX environment. The use of non-existing grains to force command sequences to be executed is a popular and useful feature of MAKE. The functionality provided by these target grains is an example of how construction tools can be used for more than just system construction.

```
M=/USR/SHC/LIB/MIP
CFLAGS=-O -DFLEXNAMES
LINTLIBS=LLIB-PORT.LN LLIB-LC.LN LLIB-LM.LN LLIB-LMP.LN LLIB-LCURSES.LN

ALL:    LPASS1 LPASS2 $(LINTLIBS)

LPASS1: CGRAM.O XDEFS.O SCAN.O COMM1.O PFTN.O TREES.O OPTIM.O LINT.O HASH.O
        CC CGRAM.O XDEFS.O SCAN.O COMM1.O PFTN.O TREES.O OPTIM.O LINT.O HASH.O -O LPASS1

TREES.O: $(M)/MANIFEST MACDEFS $(M)/MFILE1 $(M)/TREES.C
        CC -C $(CFLAGS) -I$(M) -I. $(M)/TREES.C
OPTIM.O: $(M)/MANIFEST MACDEFS $(M)/MFILE1 $(M)/OPTIM.C
        CC -C $(CFLAGS) -I$(M) -I. $(M)/OPTIM.C
PFTN.O: $(M)/MANIFEST MACDEFS $(M)/MFILE1 $(M)/PFTN.C
        CC -C $(CFLAGS) -I$(M) -I. $(M)/PFTN.C
LINT.O: $(M)/MANIFEST MACDEFS $(M)/MFILE1 LMANIFEST
        CC -C $(CFLAGS) -I$(M) -I. LINT.C
SCAN.O: $(M)/MANIFEST MACDEFS $(M)/MFILE1 $(M)/SCAN.C
        CC -C $(CFLAGS) -I$(M) -I. $(M)/SCAN.C
XDEFS.O: $(M)/MANIFEST $(M)/MFILE1 MACDEFS $(M)/XDEFS.C
        CC -C $(CFLAGS) -I$(M) -I. $(M)/XDEFS.C
COMM1.O: $(M)/MANIFEST $(M)/MFILE1 $(M)/COMMON MACDEFS $(M)/COMM1.C
        CC -C $(CFLAGS) -I. -I$(M) $(M)/COMM1.C
CGRAM.O: $(M)/MANIFEST $(M)/MFILE1 MACDEFS CGRAM.C
        CC -C $(CFLAGS) -I$(M) -I. CGRAM.C

CGRAM.C: $(M)/CGRAM.Y
        YACC $(M)/CGRAM.Y
        MV Y.TAB.C CGRAM.C

LLIB-PORT.LN: LLIB-PORT LPASS1
        -(/LIB/CPP -C -DLINT LLIB-PORT | ./LPASS1 -PUV > LLIB-PORT.LN )
LLIB-LM.LN: LLIB-LM LPASS1
        -(/LIB/CPP -C -DLINT LLIB-LM | ./LPASS1 -PUV > LLIB-LM.LN )
LLIB-LMP.LN: LLIB-LMP LPASS1
        -(/LIB/CPP -C -DLINT LLIB-LMP | ./LPASS1 -PUV > LLIB-LMP.LN )
LLIB-LC.LN: LLIB-LC LPASS1
        -(/LIB/CPP -C -DLINT LLIB-LC | ./LPASS1 -V > LLIB-LC.LN )
LLIB-LCURSES.LN: LLIB-LCURSES LPASS1
        -(/LIB/CPP -C -DLINT LLIB-LCURSES | ./LPASS1 -V > LLIB-LCURSES.LN )

LPASS2: LPASS2.O HASH.O
        CC LPASS2.O HASH.O -O LPASS2

LPASS2.O: $(M)/MANIFEST LMANIFEST
        CC $(CFLAGS) -C -I$(M) -I. LPASS2.C

LINTALL:
        LINT -HPV -I. -I$(M)  $(M)/CGRAM.C $(M)/XDEFS.C $(M)/SCAN.C \
                $(M)/PFTN.C $(M)/TREES.C $(M)/OPTIM.C LINT.C
INSTALL: ALL SHELL
        INSTALL -S LPASS1 /USR/LIB/LINT/LINT1
        INSTALL -S LPASS2 /USR/LIB/LINT/LINT2
        FOR I IN LLIB-*; DO INSTALL -C -M 644 $$I /USR/LIB/LINT; DONE
        INSTALL -C SHELL /USR/BIN/LINT
SHRINK:
        RM -F *.O

CLEAN: SHRINK
        RM -F LPASS1 LPASS2 CGRAM.C ${LINTLIBS}
```

Figure 2-4: MakeFile For LINT

Deficiencies

Phrased in terms of construction. The fundamental problem with MAKE is that it forces users to manipulate lists of construction directives. People do not normally think about systems in terms of the steps used to construct them, and therefore these lists are difficult to understand. MAKE should present a more natural user interface and then work from the user supplied information towards the construction information that it requires.

MAKE does not include an adequate means for saving and reusing common construction patterns. The introduction of such a facility would shorten MakeFiles since common patterns would be replaced with single identifiers. The definition of the identifier would document and highlight the intended construction pattern. The functionality described in this paragraph is usually provided by a macro mechanism, however the MAKE macro facility is too simple -- it does not even allow for parameterized macros.

No underlying task descriptions. Systems that keep knowledge about construction separate from knowledge about systems can be extended by adding to the construction knowledge without altering existing system models. Pitman [Pitman 84] discusses the importance of separating knowledge about systems from knowledge about construction tasks. MAKE does not use task descriptions at all and cannot be extended without changing existing MakeFiles.

Intermediate grains are referenced. Maintainers can only change systems by manipulating source grains or requesting that goal grains be constructed. Maintainers do not manipulate intermediate grains and it would be nice if these grains did not need to appear in MakeFiles.

All source grains need not be referenced. MAKE allows system descriptions to omit source grains that are also goal grains since there is no command sequence that uses or effects them. For example, there is nothing that forces UNIX Shell Scripts to be included in MakeFiles. The absence of references to Shell Scripts would be a serious omission if someone were using a MakeFile to determine which grains needed to be copied when transporting a system.

2.2 DEFSYSTEM

DEFSYSTEM [Weinreb and Moon 81] is a construction directive based tool that is used to install and maintain Lisp Machine software. The DEFSYSTEM analog to MAKE's Makefile is called a system description. DEFSYSTEM system descriptions contain a mixture of system modeling information and construction directives. DEFSYSTEM requires that command sequences (called transformations) be formally defined before they are used; this is different from the MAKE approach of allowing unlimited use of UNIX command sequences.

System descriptions are made by DEFSYSTEM macro. Calls to DEFSYSTEM have the form:

```
(DEFSYSTEM SYSTEM-NAME
  (KEYWORD ARGS ...)
  (KEYWORD ARGS ...)
  ...)
```

The options selected by the keywords fall into two general categories: properties of the system and transformations.

There are three main DEFSYSTEM property keywords:

:NAME Specifies a "pretty" version of *SYSTEM-NAME* for use in printing.

:PATHNAME-DEFAULT
Specifies a local default within the definition of the system for strings to be parsed into pathnames.

:MODULE Assigns a name to a group of files within the system.

A transformation is an operation, such as compiling or loading, that takes one or more files and performs some operation on them. There are two types of DEFSYSTEM transformations: simple and complex. A simple transformation is a single operation on a module, such as compiling it or loading it. A complex transformation combines several transformations; for example, compiling and then loading the results of the compilation.

The general format of a simple transformation is:

(NAME INPUT PRE-CONDITIONS)

NAME The name of the transformation to be performed on the files specified by *INPUT*. Examples of transformation names are :FASLOAD and :COMPILE-LOAD-INIT (these transformations are described below).

INPUT A module or nested transformation.

PRE-CONDITIONS
Optional. Specifies transformations that must occur before the current transformation itself can take place. The format is either a list *(NAME MODULE-NAMES ...)*, or a list of such lists. Each of these lists declares that the transformation *NAME* must be performed on the named modules before the current transformation can take place. (The Lisp Machine documentation calls pre-conditions *dependencies*.)

The following simple transformations are pre-defined:

`:FASLOAD` Loads the indicated file when a newer version of the file exists than was read into the current environment.

`:COMPILE` Compiles the indicated file when the source file has been been updated since the compiled code file was written.

Unlike simple transformations, complex transformations do not have any standard form. The pre-defined complex transformations are:

`:COMPILE-LOAD`
Compiles and then loads the input files. It has the form:

```
(:COMPILE-LOAD INPUT COMPILE-CONDITIONS LOAD-CONDITIONS)
```

and is exactly the same as

```
(:FASLOAD (:COMPILE INPUT COMPILE-CONDITIONS) LOAD-CONDITIONS)
```

`:COMPILE-LOAD-INIT`
Compiles and loads the input files. This transformation is sensitive to changes made to an additional dependency list. It has the form:

```
(:COMPILE-LOAD-INIT INPUT ADDITIONAL-DEPENDENCIES
 COMPILE-PRE-CONDITIONS LOAD-PRE-CONDITIONS)
```

INPUT will be compiled and loaded whenever its source file or any of the modules listed in *ADDITIONAL-DEPENDENCIES* are updated. Note, the *ADDITIONAL-DEPENDENCIES* field of this transformation specifies the same kind of construction dependency as MakeFile entries do.

It is important to distinguish between transformation declarations and transformation references. Transformations are declared by keyword lists in calls to `DEFSYSTEM`. Transformations are referenced in pre-condition lists. The transformations referenced in a pre-condition list must be declared somewhere in the system description.

DEFSYSTEM contains a facility for defining new transformations. New simple transformations are defined using the `DEFINE-SIMPLE-TRANSFORMATION` macro. Calls have the form:

```
(DEFINE-SIMPLE-TRANSFORMATION NAME FUNCTION DEFAULT-CONDITION
                              INPUT-FILE-TYPES OUTPUT-FILE-TYPES)
```

NAME The name of the transformation being defined.

FUNCTION A function to be called when the transformation is performed.

DEFAULT-CONDITION
The function that is called in order to determine if the transformation should be performed.

INPUT-FILE-TYPES
Specifies the types of the input files to the transformation. Lisp Machine file type specifications are filename extensions (e.g., "lisp" or "bin").

OUTPUT-FILE-TYPES
Specifies the types of the output files produced by the transformation.

For example, to define a simple transformation called `:LISP-YACC` that calls `LISP-YACC` to derive parsers written in Lisp from BNF grammars, the following definition could be made. (If a utility like YACC were

desired on the Lisp Machine it would probably be implemented with a macro and not a separate parser generating tool.)

```
(DEFINE-SIMPLE-TRANSFORMATION :LISP-YACC #'LISP-YACC
  #'FILE-NEWER-THAN-FILE-P (:GRAMMAR) (:LISP))
```

LISP-YACC will be invoked whenever the input file (i.e., the grammar) is newer than the output file (i.e., the parser). In other words, the transformation will be performed whenever the source file is updated. Notice that this transformation relies on grain creation dates in exactly the same way that MAKE does.

Complex transformations are defined as Lisp macros. Here is the definition of the :COMPILE-LOAD transformation that was described earlier:

```
(DEFMACRO (:COMPILE-LOAD DEFSYSTEM-MACRO)
          (INPUT &OPTIONAL COMPILE-PRE-CONDITIONS LOAD-PRE-CONDITIONS)
  '(:FASLOAD (:COMPILE ,INPUT ,COMPILE-PRE-CONDITIONS)
    ,LOAD-PRE-CONDITIONS))
```

A Small Example -- TINYCOMP

Figure 2-5 contains the DEFSYSTEM description for a Lisp implementation of TINYCOMP.

```
(DEFSYSTEM TINYCOMP
  (:MODULE DEFS "DEFINITIONS")
  (:MODULE PARSER "PARSER")
  (:MODULE CODE-GENERATOR "CODEGEN")
  (:MODULE LIBRARY "LIBRARY")

  (:FASLOAD DEFS)
  (:FASLOAD LIBRARY)
  (:COMPILE-LOAD-INIT CODE-GENERATOR  (DEFS) (:FASLOAD DEFS))
  (:COMPILE-LOAD-INIT (:LISP-YACC PARSER) (DEFS) (:FASLOAD DEFS)))
```

Figure 2-5: DEFSYSTEM Description For TINYCOMP

The TINYCOMP description contains a set of module definitions followed by a series of transformations. The transformations in the description have the following interpretation:

`(:FASLOAD DEFS)`
Specifies that DEFS should be loaded whenever it is updated. There are no pre-conditions to be satisfied before the loading can take place.

`(:FASLOAD LIBRARY)`
Specifies that LIBRARY should be loaded whenever it is updated. There are no pre-conditions to be satisfied before the loading can take place.

`(:COMPILE-LOAD-INIT CODE-GENERATOR (DEFS) (:FASLOAD DEFS))`
Specifies that CODE-GENERATOR should be be compiled and loaded whenever it or DEFS changes. Before the compilation can take place, DEFS must be loaded.

`(:COMPILE-LOAD-INIT (:LISP-YACC PARSER) (DEFS) (:FASLOAD DEFS))`
Specifies that a parser derived from PARSER is to be compiled and loaded. A new parser is produced whenever PARSER changes. The compiler and loader are invoked whenever DEFS or the derived parser changes. :LISP-YACC will not be invoked if only DEFS changes. Prior to compilation, DEFS must be loaded.

The Construction Process

Systems previously modeled with DEFSYSTEM are constructed by calling MAKE-SYSTEM. Calls have the form:

```
(MAKE-SYSTEM SYSTEM-NAME &REST OPTIONS)
```

SYSTEM-NAME Specifies a system previously modeled with DEFSYSTEM.

OPTIONS Specifies options like *print the transformations that would be done but don't do them* and so forth.

The construction dependency graph specified by the transformations and pre-conditions in the DEFSYSTEM description of *SYSTEM-NAME* is analyzed in order to determine what construction needs to be done. Each transformation is applied by first applying any transformations referenced as pre-conditions, and then updating the input module if it, or any modules listed in additional dependency lists, have been changed. Notice that the transformation applications are ordered by the pre-condition lists.

Like MAKE, DEFSYSTEM uses simple functions based on file creation dates in order to determine when a module should be reconstructed. However, unlike MAKE, DEFSYSTEM allows the optional specification of predicates that control when construction is done. The new predicates can replace the simple ones that are supplied with DEFSYSTEM.

DEFSYSTEM includes a patching facility. It allows small changes to be made to a system without invoking the DEFSYSTEM transformation/dependency mechanism. Each set of changes is stored in a patch file that typically contains new function definitions or redefinitions of old functions. Each patch is assigned a number. If a system contains patches, then the patches are loaded, in order, after the unpatched version of the system is loaded.

An Extended Example -- LINT

The DEFSYSTEM description for a Lisp implementation of LINT is presented in figure 2-6. Although the DEFSYSTEM description is easier to understand than the corresponding MakeFile (figure 2-4), it is still difficult to understand.

The :BUILD-LINT-LIBRARY transformation is assumed to have been defined and has the form:

```
(:BUILD-LINT-LIBRARY INPUT PRE-CONDITIONS)
```

It constructs LINT library files from LINT library sources. The transformation allows the optional specification of pre-conditions, and is applied if either *INPUT*, or the first pass of LINT is updated.

The first keyword form in the LINT DEFSYSTEM description specifies a system-wide default directory. The next block of keyword forms declare the various modules which comprise LINT. The final block of forms declare the transformations used to construct LINT. Notice that as transformations are nested and pre-conditions are added, the transformation declarations become increasingly difficult to understand.

Deficiencies

Phrased in terms of construction. Like MAKE, DEFSYSTEM is a construction directive based tool. This is the primary reason that DEFSYSTEM descriptions, although easier to understand than MakeFiles, are still awkward.

One reason that DEFSYSTEM descriptions are easier to understand than MakeFiles is because DEFSYSTEM is not purely construction directive based. DEFSYSTEM's :MODULE declarations allow for the logical grouping of grains into higher level modules. This grouping abstracts away from low level construction information, and provides a more natural way for users to describe systems than MAKE does.

DEFSYSTEM supports the sharing of common construction patterns through the declaration of

```
(DEFSYSTEM LINT
  (:PATHNAME-DEFAULT "/USR/SRC/LIB/MIP")
  (:MODULE DEFINITIONS-1 ("MACDEFS" "MANIFEST" "MFILE1" "LMANIFEST"))
  (:MODULE DEFINITIONS-2 ("MANIFEST" "LMANIFEST"))
  (:MODULE PARSER "CGRAM")
  (:MODULE PASS1 ("XDEFS" "SCAN" "COMM1" "PFTN" "TREES" "OPTIM"
                  "LINT" "HASH"))
  (:MODULE PASS2 ("LPASS2" "HASH"))
  (:MODULE DRIVER "SHELL")
  (:MODULE LIBRARIES ("LLIB-PORT" "LLIB-LC" "LLIB-LM" "LLIB-LMP"
                      "LLIB-LCURSES"))

  (:FASLOAD DEFINITIONS-1)
  (:FASLOAD DEFINITIONS-2)
  (:COMPILE-LOAD DRIVER)
  (:COMPILE-LOAD-INIT PASS1 (DEFINITIONS-1) (:FASLOAD DEFINITIONS-1))
  (:COMPILE-LOAD-INIT PASS2 (DEFINITIONS-2) (:FASLOAD DEFINITIONS-2))
  (:COMPILE-LOAD-INIT (:LISP-YACC PARSER) (DEFINITIONS-1)
                      (:FASLOAD DEFINITIONS-1))
  (:BUILD-LINT-LIBRARY LIBRARIES (:FASLOAD DRIVER PARSER PASS1)))
```

Figure 2-6: DEFSYSTEM Description For LINT

transformations. This makes DEFSYSTEM system descriptions easier to produce and understand than MakeFiles. However, since it is possible to avoid the declaration of a complex transformation by using nested transformations, DEFSYSTEM still allows for common patterns to be repeated instead of shared.

No underlying task descriptions. Although DEFSYSTEM has embedded knowledge about Lisp compilation and loading it does not include a mechanism for describing construction tasks and therefore cannot be extended without great difficulty.

Intermediate grains are referenced. DEFSYSTEM does not differentiate between source, intermediate, and goal grains. In general, intermediate grains are hidden by complex transformations. For example, there are no references to intermediate grains in figures 2-5 and 2-6. While DEFSYSTEM does not force intermediate grains to be included, it does not prohibit them either.

All source grains need not be referenced. In a Lisp environment, nothing can be used before it is loaded. This means that any grain that participates in a Lisp system will be involved in some construction, and therefore, it is not as natural to omit a source grain from a DEFSYSTEM description as it is to omit one from a MakeFile. This difference between MAKE and DEFSYSTEM comes from differences between the UNIX and Lisp environments, and not from important differences between the two tools.

2.3 Other Tools

DeRemer and Kron introduced the terms programming-in-the-large and programming-in-the-small [DeRemer and Kron 76] to distinguish between the writing of modules and the structuring of modules into systems. Consistent construction is just one programming-in-the-large issue, others include source code management, module interconnection specification, and version control. A brief summary of these other issues and projects that focus upon them is presented here for completeness. The consistent construction components of these projects do not differ from MAKE or DEFSYSTEM in any significant way.

When several people are working on a system simultaneously, it is important to regulate access to the source code modules in order to ensure that someone does not attempt to modify a module while someone else is modifying that same module. A common scheme is to implement a *librarian* that regulates access to system components via a check-in/check-out mechanism. In short, only one person is allowed to check-out a module for update at any time. Anyone can read a module at any time. Source code management systems are described in the following papers [Rochkind 75, Cristofor, et. al. 80, Hunsley, and Lynch 79, Lewis 83].

All of the problems mentioned above are compounded if the programming environment is distributed over a network. Schmidt addresses these issues [Schmidt 82].

It is often the case that there are families of systems being managed. For example there may be several public releases of a system, internal releases, experimental versions and so on. It is also common for there to be several versions of a system intended to run on different hardware configurations. Each member of a family of software systems usually shares many components with other members of the family. Maintainers of such families need to worry about which versions of which modules are used in each member of the family. Tichy and Cooprider attacked the problems associated with the representation and management of software families [Cooprider 79, Tichy 80, Tichy 84].

3. The BUILD Reference Level

This chapter introduces BUILD's reference based system modeling scheme. BUILD system models are very easy to interpret because they contain nothing more than declarations of how grains are grouped to form modules and how these modules refer to each other. Although they do not present any construction dependencies explicitly, they can be used to derive all of the construction information found in construction based models (see Chapter 5). Construction models cannot be used to derive the reference information found in reference models. Reference models are far less confusing than the construction based models because they are written in a language that replaces low level grain construction information with higher level inter-module reference patterns.

3.1 Modules

It is often the case that groups of grains are conceived as one logical entity but are split up (e.g. into files) for other reasons. Modeling schemes that represent systems only at the level of the individual grain do not have the ability to express this kind of grouping. The module construct used by BUILD (and DEFSYSTEM) allows these groupings to be made explicitly in system descriptions.

BUILD module declarations have the form:

```
(:MODULE MODULE-NAME GRAIN-TYPE &REST GRAINS)
```

MODULE-NAME The name of a module. The name must be unique within the system model.

GRAIN-TYPE The name of a grain type recognized by BUILD. Each grain is assumed to be an instance of this type.

GRAINS The names of the grains that comprise the module.

The following form declares that `MAIN` is a Lisp source module composed of the single grain `MAIN.LISP`,

```
(:MODULE MAIN :LISP-SOURCE "MAIN.LISP")
```

and the form:

```
(:MODULE DEFS :C-SOURCE "DEFINITIONS-1.C" "DEFINITIONS-2.C")
```

declares that `DEFS` is a C source module with two grains named `DEFINITIONS-1.C` and `DEFINITIONS-2.C`.

BUILD can use grain type information without considering module references to determine a great deal about the construction of grains. For instance, BUILD knows how to invoke the correct compiler on C or Lisp source files or how to construct LINT library files from library sources by utilizing grain type information alone.

3.2 References

BUILD infers construction dependencies from reference assertions by taking advantage of the fact that construction dependencies are caused by references between modules. If two modules do not refer to each other, then it is impossible for there to be a construction dependency that involves them. When the assertion is made that $module_1$ refers to $module_2$, BUILD pessimistically assumes that each grain in $module_1$ refers to each grain in $module_2$.

References with the same name may be handled differently depending upon the grain types of the modules involved in the reference. For instance, the *calls* reference between two Lisp source modules is handled differently than the *calls* reference between two C source modules.

BUILD reference declarations provide for the specification of references between modules. No meaning is attached to the ordering of reference declarations. Reference declarations have the form:

(REFERENCE LEFT-ELEMENT RIGHT-ELEMENT)

REFERENCE The name of a reference recognized by BUILD.

LEFT-ELEMENT A module name or list of module names. All module names used in a reference declaration must have been declared in a module declaration.

RIGHT-ELEMENT
A module name or list of module names. All module names used in a reference declaration must have been declared in a module declaration.

The use of module name lists as either of the elements of a reference declaration is syntactic sugar that is equivalent to the set of reference declarations composed by enumerating *REFERENCE-NAME* with each pair in the cross product of the right and left element lists. For example:

```
(:CALLS (A B) (D E))
```

is equivalent to:

```
(:CALLS A D)
(:CALLS A E)
(:CALLS B D)
(:CALLS B E)
```

Here are some reference triples and the construction dependencies that they imply:

`(:CALLS LISP-SOURCE-1 LISP-SOURCE-2)`
Asserts that `LISP-SOURCE-1` contains functions that call `LISP-SOURCE-2` and implies that `LISP-SOURCE-2` will need to be loaded in order for `LISP-SOURCE-1` to execute.

`(:MACRO-CALLS LISP-SOURCE-1 LISP-SOURCE-2)`
Asserts that `LISP-SOURCE-1` uses macros defined in `LISP-SOURCE-2` and therefore `LISP-SOURCE-2` must be loaded in order for `LISP-SOURCE-1` to compile properly. This reference implies that if `LISP-SOURCE-2` changes, then `LISP-SOURCE-1` will need to be re-compiled.

`(:CALLS C-SOURCE-1 C-SOURCE-2)`
Implies that the object grains compiled from `C-SOURCE-2` (as well as the object grains from any module that `C-SOURCE-2` calls) need to be linked into any executable image that is to include the object grains from `C-SOURCE-1`.

`(:INCLUDES C-SOURCE-1 C-SOURCE-2)`
Asserts that `C-SOURCE-1` contains the contents of `C-SOURCE-2`. This reference implies that whenever the included module, `C-SOURCE-2`, changes, the including module, `C-SOURCE-1`, needs to be rebuilt.

BUILD uses triples (called reference signatures) of the form
<REFERENCE-NAME LEFT-GRAIN-TYPE-NAME RIGHT-GRAIN-TYPE-NAME>
to identify references. BUILD uses grain type information to distinguish between references that have the same name but apply to different grain types. A given implementation of BUILD will define the reference signatures that are commonly used in the environment that BUILD is working with. Chapter 5 describes how new reference signatures may be added to BUILD.

3.3 Models

The general form of a BUILD system description is:

```
(DEFMODEL MODEL-NAME &REST DECLARATIONS)
```

There are four kinds of declarations that may be included in a DEFMODEL form: module, reference, default pathname, and default module. Module and reference declarations were described earlier in this chapter. The default pathname declaration allows for the declaration of a pathname to be used as a template for completing filenames. It has the form:

```
(:DEFAULT-PATHNAME PATHNAME)
```

The default module declaration is used to declare a module as the default module for BUILD to operate on when construction requests for the system are made. It has the form:

```
(:DEFAULT-MODULE MODULE-NAME)
```

Figure 3-1 contains the DEFMODEL form for TINYCOMP. The first four declarations are module declarations that specify the grains and grain types of the system modules. The last three declarations specify the references between the modules in the system. Figure 3-2 contains the DEFMODEL form for LINT. The model is longer than the TINYCOMP model but no more complicated.

```
(DEFMODEL TINYCOMP
  (:MODULE DEFS :C-SOURCE "DEFINITIONS")
  (:MODULE PARSER :YACC-GRAMMAR "PARSER")
  (:MODULE CODE-GENERATOR :C-SOURCE "CODEGEN")
  (:MODULE LIBRARY :C-OBJECT "LIBRARY")

  (:INCLUDES (PARSER CODE-GENERATOR) DEFS)
  (:CALLS PARSER (LIBRARY CODE-GENERATOR))
  (:CALLS CODE-GENERATOR LIBRARY))
```

Figure 3-1: BUILD Model For TINYCOMP

```
(DEFMODEL LINT
  (:DEFAULT-PATHNAME "/USR/SRC/LIB/MIP")
  (:MODULE DEFINITIONS-1 :C-SOURCE
   "MACDEFS" "MANIFEST" "MFILE1" "LMANIFEST")
  (:MODULE DEFINITIONS-2 :C-SOURCE "MANIFEST" "LMANIFEST")
  (:MODULE PARSER :GRAMMAR "CGRAM")
  (:MODULE PASS-1 :C-SOURCE "LINT")
  (:MODULE PASS-2 :C-SOURCE "LPASS2")
  (:MODULE SUPPORT-1 :C-SOURCE
   "XDEFS" "SCAN" "COMM1" "PFTN" "TREES" "OPTIM" "HASH")
  (:MODULE SUPPORT-2 :C-SOURCE "HASH")
  (:MODULE DRIVER :SHELL-SCRIPT "SHELL")
  (:MODULE LIBRARIES :LINT-LIBRARY-SOURCE
   "LLIB-PORT" "LLIB-LC" "LLIB-LM" "LLIB-LMP" "LLIB-LCURSES")

  (:INCLUDES PASS-1 DEFINITIONS-1)
  (:INCLUDES PASS-2 DEFINITIONS-2)
  (:CALLS DRIVER (PASS-1 PASS-2 LIBRARIES))
  (:CALLS PASS-1 (PARSER SUPPORT-1))
  (:CALLS PASS-2 SUPPORT-2))
```

Figure 3-2: BUILD Description For LINT

4. The BUILD Task Level

This chapter describes the task level representation of systems used by BUILD. A task level model is derived from the reference level model for each request that BUILD receives. The derived model is then used to handle the request. (The phrase *task level* is used in place of the more specific phrase *construction level* because BUILD is used for more than just construction.)

BUILD task level models are acyclic directed graphs with two kinds of nodes: *g-nodes* which represent grains, and *p-nodes* which represent the processes used to construct grains. Leaf nodes represent source grains, and root nodes represent goal grains. The link between grains and the processes that use them is modeled by linking the g-nodes representing grains to the p-nodes representing the processes that use them.

Figure 4-1 contains a portion of the task graph used to represent the compilation of PARSER.LISP, a grain from a Lisp implementation of TINYCOMP. This example assumes that PARSER.LISP is a source grain and ignores the fact that in TINYCOMP, PARSER.LISP is an intermediate module produced by LISP-YACC. The ellipses represent g-nodes and the rectangles represent p-nodes. There are two source nodes, PARSER.LISP and DEFS.LISP, and a single goal node, PARSER.IMAGE.

Although the use of an acyclic directed graph to represent task processing is not unique (MAKE and DEFSYSTEM use similar representations) the derivation of task graphs from reference models is novel.

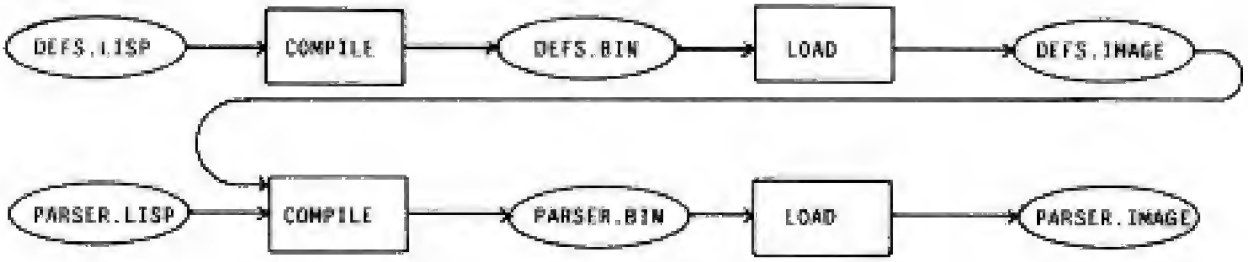

Figure 4-1: Simple Task Graph

4.1 Grain types

Grain type objects are used to represent the classes of grains used by the environment that BUILD is working with. They are used to represent all of the kinds of grains that are manipulated by the underlying environment, whether they are files or not. For instance, the grain type :LISP-IMAGE is used to represent the objects that result from loading files into the Lisp environment.

Defining Grain Types

Grain types are defined with DEFINE-GRAIN-TYPE and definitions have the form:

(DEFINE-GRAIN-TYPE *NAME* &OPTIONAL *FILENAME-EXTENSION*)

NAME The name of the grain type being defined.

FILENAME-EXTENSION
The default filename extension for grains of this type. If this field is null then BUILD assumes that grains of this type are not files.

Figure 4-2 contains the grain type definitions used to model Lisp systems. The :LISP-SOURCE and :LISP-BINARY grain types correspond to files and hence their definitions include default filename extensions (the Lisp Machine uses keyword symbols to represent filename extensions). The :LISP-IMAGE grain type is not associated with files and therefore has no default filename extension.

```
(DEFINE-GRAIN-TYPE :LISP-SOURCE :LISP)
(DEFINE-GRAIN-TYPE :LISP-BINARY :BIN)
(DEFINE-GRAIN-TYPE :LISP-IMAGE)
```

Figure 4-2: Grain Type Definitions for Lisp

4.2 G-nodes

G-nodes represent grains in task graphs, they contain the following information:

NAME The name of the grain represented by this g-node.

TYPE The grain type object that the grain represented by this g-node is an instance of.

MODULE Optional. The module that includes the grain represented by this g-node.

CREATOR Optional. The p-node that represents the process that created this g-node. This field will be null if the g-node represents a source grain.

USERS A list of p-nodes that depend on this g-node to fill an input role.

INGREDIENTS A list that represents the source grains used to produce this g-node. Each element of the list is a pair containing the name and creation-date of an ingredient grain.

CREATE-DATE A time stamp that represents the time and date when the grain that is represented by this g-node was created.

4.3 Process Types

Process type objects contain the information pertaining to classes of process instances (represented by p-nodes). For example, the Lisp Machine implementation of BUILD includes process type objects for Lisp compilation and Lisp binary file loading.

The grains that are used and produced by processes are partitioned according to the roles that they play in them. Grains that processes use are said to play *input roles*. Grains that are produced by processes are said to play *output roles*.

Process type objects contain role descriptions for each of their input and output roles. Role descriptions contain the following information:

NAME The name of the role. It must be unique within the process type being defined.

GRAIN-TYPE The grain type name that grains filling this role must have.

ARITY Either :SINGLE or :MULTIPLE. A role with arity :SINGLE can have no more than one grain filling it. A role with arity :MULTIPLE can have an arbitrary number of grains filling it.

NAME-SOURCE Optional. The name of a role used to help derive names for grains that will fill this role.

Defining Process Types

Process types are defined with DEFINE-PROCESS-TYPE and calls have the form:

```
(DEFINE-PROCESS-TYPE NAME INPUT-SPEC OUTPUT-SPEC STREAM-VAR
                     DESCRIBE-FORM &REST CONSTRUCT-FORMS)
```

NAME The name of the process type.

INPUT-SPEC A list of input role descriptions (discussed above).

OUTPUT-SPEC A list of output role descriptions.

STREAM-VAR A variable name that will be bound to the output stream when *DESCRIBE-FORM* and *CONSTRUCT-FORMS* are evaluated.

DESCRIBE-FORM

A form to be evaluated in order to describe the processing represented by an instance of this process type. When the form is evaluated, each role-name will be bound to the names of the grains playing the role. Also, the symbol named by *STREAM-VAR* will be bound to the output stream.

CONSTRUCT-FORMS

The forms to be evaluated in order to accomplish the processing represented by an instance of the process type. When these forms are evaluated each of the role-names and the symbol named by *STREAM-VAR* will be bound as mentioned above.

Figure 4-3 contains the process type definitions for Lisp compilation and Lisp binary loading. The definition for :LISP-COMPILE specifies that there are two input roles, SOURCE and DEFINITIONS, and a single output role, BINARY. SOURCE has singular arity and must be filled by a :LISP-SOURCE grain. DEFINITIONS has multiple arity and can only be filled by :LISP-IMAGE grains. BINARY has singular arity and must be filled by a :LISP-BINARY grain. The describe form produces descriptions like:

```
"COMPILE PARSER.LISP"
```

The construct forms produce the grain playing the BINARY role by compiling the grain playing the SOURCE role. The construct forms also cause a notification of the compilation to be sent to the output stream. The notification looks like:

```
"COMPILING PARSER.LISP.5"
```

Processes often depend on grains not explicitly mentioned in their invocations. For example, in languages that rely on objects to be specified or loaded before objects that refer to them can be compiled, the compilation process type must include a role that is used to capture that dependency. The role DEFINITIONS is used in :LISP-COMPILE in order to express the need for some things to be defined before a Lisp grain can be compiled. The link between the g-node for DEFS.IMAGE and the p-node representing the compilation of PARSER.LISP in the task model from figure 4-1 is an example of such a dependency being modeled. Another situation in which it is necessary to model a dependency not made explicitly in command line invocation is for C compilation. The :C-COMPILE process type has the role INCLUDE to represent the dependency between a file and the files that it includes via the C #INCLUDE mechanism.

```
(DEFINE-PROCESS-TYPE :LISP-COMPILE
    ((SOURCE :LISP-SOURCE :SINGLE)              ;SOURCE INPUT ROLE
     (DEFINITIONS :LISP-IMAGE :MULTIPLE))       ;DEFINITIONS INPUT ROLE
    ((BINARY :LISP-BINARY :SINGLE SOURCE))      ;BINARY OUTPUT ROLE
    OUTPUT-STREAM                               ;STREAM-VAR
    (FORMAT OUTPUT-STREAM "~%COMPILE ~A"        ;DESCRIBE-FORM
            (PATHNAME-MINUS-VERSION SOURCE))
  (FORMAT OUTPUT-STREAM "~%COMPILING ~A" SOURCE)     ;CONSTRUCT-FORMS
  (COMPILER:COMPILE-FILE SOURCE BINARY))

(DEFINE-PROCESS-TYPE :LISP-LOAD-BIN
    ((BINARY :LISP-BINARY :SINGLE)              ;BINARY INPUT ROLE
     (DEFINITIONS :LISP-IMAGE :MULTIPLE))       ;DEFINITIONS INPUT ROLE
    ((IMAGE :LISP-IMAGE :SINGLE BINARY))        ;IMAGE OUTPUT ROLE
    OUTPUT-STREAM                               ;STREAM-VAR
    (FORMAT OUTPUT-STREAM "~%LOAD ~A"           ;DESCRIBE-FORM
            (PATHNAME-MINUS-VERSION BINARY))
    (FORMAT OUTPUT-STREAM "~%LOADING ~A" BINARY)     ;CONSTRUCT-FORMS
  (SI:LOAD-BINARY-FILE BINARY NIL T))
```

Figure 4-3: Process Type Definitions For Lisp

4.4 P-Nodes

Each p-node represents a process to be invoked on the grains attached to its input ports to produce the grains attached to its output ports. Each role in a process type is represented as a port in p-nodes of that type. The grain type of each g-node attached to a port must be the same as the grain type associated with the role. A description of the processing represented by a p-node and the g-nodes attached to its ports can be produced by applying *DESCRIBE-FORM* from the p-node's process type object to the p-node. The processing represented by the p-node can be done by applying *CONSTRUCT-FORMS* from the p-node's process type object to the p-node.

Figure 4-4 contains an expanded view of the p-node used to represent the compilation of PARSER.LISP in TINYCOMP.

Figure 4-4: Expanded P-Node

4.5 Task Graph Constraints

Task graphs are constrained in the following ways:

1. Task graphs are acyclic. A cycle in a graph would imply that some grain was needed in order to construct itself.

2. The parent of a g-node, if there is one, must be a p-node.

3. A g-node can have no more than one parent.

4. A g-node without a parent represents a source grain.

5. The children of a g-node, if there are any, must be p-nodes. These nodes represent processes that depend upon the grain represented by the g-node.

6. A g-node without children represents a goal grain.

7. The children of a p-node must be g-nodes. These g-nodes represent grains derived by the process represented by the p-node. Each p-node must have at least one child.

In other words, task graphs are acyclic graphs which begin with g-nodes that represent source grains and end with g-nodes that represent goal grains. The g-nodes are separated by p-nodes that represent the processes that derive later g-nodes from earlier ones.

Figures 1-2, 2-1, and 4-1 are examples of well formed task graphs.

4.6 The Construction Algorithm

Figure 4-5 contains the algorithm used by BUILD to perform the construction modeled by a task graph. This algorithm is similar to the one used by MAKE and DEFSYSTEM (figure 2-3), the primary difference between the two algorithms is in how they make use of creation dates to determine when construction is necessary. The MAKE algorithm uses file creation date ordering between input and output grains in order to infer that an input has changed (and therefore construction is triggered). In practice this method works, however, it relies on several assumptions that are not necessarily true.

MAKE and DEFSYSTEM assume that files with the same name but different extensions are related. For instance, they assume that `MAIN.O` was created by compiling `MAIN.C`. While this is a reasonable assumption, it does not have to be true. Nothing prevents users from renaming files and therefore, there is no guarantee that `MAIN.O` actually came from `MAIN.C`.

If an output grain contains a file creation date that is newer than all of the input grains used to produce it, then MAKE and DEFSYSTEM assume that the output grain does not need to be rebuilt. However, there is no guarantee that file creation dates have not been tampered with.

BUILD does not use file creation date ordering to infer that an object has changed. BUILD compares a grain's ingredient list with the ingredient list that would result if the processing modeled by the task graph were done. If the ingredient lists match, then the construction is not done.

The prototype implementation of BUILD keeps a separate data file that contains grain creation dates and ingredients. Such a file would not be needed if the underlying environment recorded the ingredients used to produce an object. The Mesa environment [Mitchell 79, Schmidt 82] keeps this information and exploits it in order to determine when processing needs to be done.

```
(DEFUN CONSTRUCT-G-NODE (G-NODE)
  (COND ((SOURCE-NODE-P G-NODE) T)
        ((OR (NON-EXISTENT G-NODE) (INGREDIENTS-CHANGED G-NODE))
         (MAPCAR #'CONSTRUCT-G-NODE (INPUTS (PARENT G-NODE)))
         (DO-CONSTRUCTION (PARENT G-NODE)))))

(DEFUN INGREDIENTS-CHANGED (G-NODE)
  (NOT (EQUAL (INGREDIENTS G-NODE)
              (DERIVE-INGREDIENTS G-NODE))))

(DEFUN SOURCE-NODE-P (G-NODE)
  ;; RETURNS T IF AND ONLY IF G-NODE
  ;; REPRESENTS A SOURCE GRAIN
  )

(DEFUN NON-EXISTENT (G-NODE)
  ;; RETURNS T IF THE GRAIN REPRESENTED BY G-NODE
  ;; DOES NOT EXIST
  )

(DEFUN PARENT (G-NODE)
  ;; RETURN THE PARENT P-NODE OF G-NODE
  )

(DEFUN INPUTS (P-NODE)
  ;; RETURN THE INPUT G-NODES OF P-NODE
  )

(DEFUN DO-CONSTRUCTION (P-NODE)
  ;; PERFORM CONSTRUCTION REPRESENTED BY P-NODE
  )

(DEFUN INGREDIENTS (G-NODE)
  ;; RETURN THE INGREDIENT LIST USED TO CONSTRUCT
  ;; THE EXISTING VERSION OF G-NODE
  )

(DEFUN DERIVE-INGREDIENTS (G-NODE)
  ;; RETURN THE INGREDIENT LIST THAT WOULD RESULT IF
  ;; A NEW VERSION OF G-NODE WERE CONSTRUCTED
  )
```

Figure 4-5: BUILD Construction Algorithm

5. Construction Requests and Task Graph Derivation

After a system has been modeled with DEFMODEL, BUILD can be called upon to handle construction requests for it. Each request has the form:

```
(BUILD-REQUEST MODEL REQUEST &OPTIONAL MODULE (MODE :NORMAL))
```

MODEL The name of a model previously defined with DEFMODEL.

REQUEST The name of a request recognized by BUILD (e.g. :COMPILE, :LOAD).

MODULE The name of a module to operate upon. If this field is not specified then the default module for the system (as defined with the :DEFAULT-MODULE declaration form) is used.

MODE Specifies one of several construction modes. Construction modes are discussed below.

The prototype implementation of BUILD has three construction modes that behave as follows:

:NORMAL Describe all of the construction to be done, and then ask the user if BUILD should perform the construction just described.

:DESCRIBE Describe all of the construction to be done but do not perform it.

:NO-CONFIRM Perform the required construction without describing it first.

Sample BUILD requests are:

```
(BUILD-REQUEST TINY-COMP :LOAD)
(BUILD-REQUEST LINT :LOAD DRIVER)
(BUILD-REQUEST LINT :LOAD DRIVER :DESCRIBE)
```

Once a request has been received, a three step process is executed for each grain in the module stated in the request. This process creates a task model for the request which is then processed in the manner outlined in chapter 4. The three steps are:

1. Model the construction that can be deduced from the request without considering any references. This phase is called *pre-reference request processing.*

2. Model the construction that is implied by the references that involve the module associated with the request. This phase is called *reference processing.*

3. Model the construction that can be deduced from the request and the graph built from the earlier steps. This phase is called *post-reference request processing.*

After the post-reference processing has been completed the task graph is complete and can be used to direct the construction needed to handle the request.

Before the construction process can be explained in detail it is necessary to present the functions used to view and manipulate task graphs.

5.1 Viewing and Manipulating Task Graphs -- ACCESS

Consider the following task graph:

Starting at a p-node, the path to any of the g-nodes connected to one of its ports can be specified by mentioning the name of the port desired. In the task graph above, starting at the `:LISP-COMPILE` p-node, the *step* `BINARY` leads to `DEFS.BIN`.

A step from a g-node to a p-node can be described by specifying the process type of the connected p-node and the role played by the g-node in the p-node. In the sample task graph above, the step `(BINARY :LISP-COMPILE)` starting at `DEFS.BIN` leads to the `:LISP-COMPILE` p-node.

Paths are formed by listing steps:

* The path `((SOURCE :LISP-COMPILE) BINARY)` starting at `DEFS.LISP` leads to `DEFS.BIN`.

* The path `((SOURCE :LISP-COMPILE) BINARY (BINARY :LISP-LOAD-BIN))` starting at `DEFS.LISP` leads to the `:LISP-LOAD-BIN` p-node.

* The path
  `((SOURCE :LISP-COMPILE) BINARY (BINARY :LISP-LOAD) IMAGE)`
  starting at `DEFS.LISP` leads to `DEFS.IMAGE`.

* The path
  `((IMAGE :LISP-LOAD) BINARY (BINARY :LISP-COMPILE) SOURCE)`
  starting at `DEFS.IMAGE` leads to `DEFS.LISP`.

The `ACCESS` family of functions are designed to provide a straightforward mechanism for both viewing and manipulating task graphs. These functions are used heavily during the task graph derivation process. There are three functions, `ACCESS`, `ACCESS+`, and `ACCESS*`, each of which is `SETF`able. The `ACCESS` functions have the form:

(FUNCTION NODE PATH)

FUNCTION — `ACCESS`, `ACCESS+`, or `ACCESS*`.

NODE — Either a p-node or a g-node. This node is used as the root of the path to be traced by *ACCESS-FUNCTION*.

PATH — A list of steps to be traced from *NODE*.

The functions behave in the following manner:

`ACCESS` — Traces *PATH* from *NODE* and returns the last node encountered. An error is signalled if any step in *PATH* cannot be traced. An error is signalled if there could be more than one node that satisfies the path traced.

ACCESS+ Traces *PATH* from *NODE* and returns a list of nodes that satisfy the path. An error is signalled if any step in *PATH* cannot be traced.

ACCESS* Traces *PATH* from *NODE* and returns the single node that satisfies the path. An error is signalled if there could be more than one node that satisfies *PATH*. New nodes are created if steps in *PATH* do not exist.

Any ACCESS call that returns a single node may be used to specify the root of another call to ACCESS, in other words, the following two calls are equivalent:

```
(ACCESS NODE (STEP1 STEP2 STEP3))
(ACCESS (ACCESS (ACCESS NODE STEP1) STEP2) STEP3)
```

Each of the ACCESS functions can be SETFed. Calls have the form:

(SETF (ACCESS *ROOT-NODE PATH*) *END-NODE*)
Ensures that future calls to ACCESS with *ROOT-NODE* and *PATH* (i.e., (ACCESS *ROOT-NODE PATH*)) will return *END-NODE*.

(SETF (ACCESS+ *ROOT-NODE PATH*) *NODE-LIST*)
Ensures that future calls to ACCESS+ with *ROOT-NODE* and *PATH* (i.e., (ACCESS+ *ROOT-NODE PATH*)) will return *NODE-LIST*.

(SETF (ACCESS* *ROOT-NODE PATH*) *END-NODE*)
Ensures that future calls to ACCESS* with *ROOT-NODE* and *PATH* (i.e., (ACCESS* *ROOT-NODE PATH*)) will return *END-NODE*.

The ACCESS functions differ in how they handle steps that cannot be traced, and what they do when a path description fans out. If ACCESS or ACCESS+ encounter a missing link, an error is signalled. ACCESS* and the SETF functions will create the link and continue tracing the path.

A fanout condition occurs when an attempt it made to trace from a :MULTIPLE arity port of a p-node, or when more than one p-node satisfies the role-name/process-type-name constraint tracing from a g-node. ACCESS, ACCESS* and their associated SETF functions signal errors if fanout is encountered. ACCESS+ will continue tracing down all paths and returns the list of nodes that satisfied the path description. When SETFed, ACCESS+ will signal an error if fanout is encountered before the last step in the path description.

In Lisp Machine Lisp [Weinreb and Moon 81] and Common Lisp [Steele 84], the special form PUSH can be used for functions that are SETFable. PUSH can be used to add a g-node to a port. For example:

```
(PUSH SOME-G-NODE (ACCESS+ P-NODE SOME-PATH))
```

is equivalent to:

```
(SETF (ACCESS+ P-NODE SOME-PATH)
      (CONS SOME-G-NODE (ACCESS+ P-NODE SOME-PATH)))
```

The SETF forms and ACCESS* can make additive changes to the graph. When a function needs to create a g-node and link it to a p-node port, a name needs to be synthesized for the new g-node. The name of each g-node resembles a filename in that it has two parts, a primary name and an extension. In order to synthesize a g-node name, the function copies the primary part from the grain attached to the port specified as the *NAME-SOURCE* port for the port being linked to (see the paragraph about role descriptions in chapter 4). An error is signalled if a function needs to derive a g-node name to link to a port that has no *NAME-SOURCE* port associated with it. The extension of a g-node name is derived from its grain type object. If the grain type represents files, then the extension is the default-filename-extension, otherwise, it is the name of the grain type itself.

5.2 Request Handlers

Request handlers specify the task graph derivation steps that can be taken whenever the request associated with the handler has been made, without considering any reference declarations. Requests are identified with request signatures (much like reference signatures). Each request signature contains two fields, a request name and a grain type name. For example the signature:

```
<:COMPILE :LISP-SOURCE>
```

identifies the handler designed to build part of the task graph needed to accomplish the compilation of a Lisp source grain. The signature:

```
<:YACC :YACC-GRAMMAR>
```

identifies the handler that will build part of the task graph needed to invoke YACC on a grammar.

Not all possible signatures will have handlers defined for them. For example the request signature:

```
<:COMPILE :LISP-BINARY>
```

identifies a nonsensical request.

Pre-reference request handlers are used to construct the parts of a task graph which will be needed regardless of the ramifications of references. For example, in order to model the compilation of some :LISP-SOURCE grain, G.LISP, the following links can be made without considering any references; the g-node representing G.LISP should be linked to the SOURCE port of a :LISP-COMPILE p-node, and then the BINARY port of this p-node should be linked to a g-node representing the binary version of G.LISP (i.e., G.BIN).

Post-reference request handlers are used for modeling processing that can only be deduced after the implications of the references are added to the task graph. At this time it has not been necessary to use a post reference handler, however, they are included because there may be situations where their use is appropriate.

Defining Request Handlers

Request handlers are defined with DEFINE-REQUEST-HANDLER. Calls have the form:

```
(DEFINE-REQUEST-HANDLER (REQUEST GRAIN-TYPE-NAME PRE-OR-POST)
                        (ARGS)
  &BODY BODY)
```

REQUEST The name of the request being handled.

GRAIN-TYPE-NAME
The type of the grain that the handler is for.

PRE-OR-POST :PRE indicates that this is a pre-reference handler. :POST indicates that this is a post reference handler.

ARGS The names of the variables passed to the handler. There must be at least one element in this list. The first *ARG* will be bound to the g-node associated with the request when *BODY* is evaluated.

BODY The forms that constitute the handler. They are evaluated with the arguments passed to the handler bound to the variables named in *ARGS*.

All requests made by users have a single argument, the name of the module that the request is intended for. Handlers may also make requests, and these requests can contain more than one argument. The handlers for the `:LOAD+` and `:INCLUDE+` tasks presented in Appendix I are examples of handlers using additional arguments.

Figure 5-1 contains the request handler definitions for Lisp compilation and loading. The first handler is invoked when a `:COMPILE` request is made on a `:LISP-SOURCE` module. It uses `ACCESS*` to ensure that the task graph being derived models the fact that the `:LISP-SOURCE` grains in the module need to be compiled.

The second handler is invoked when a `:LOAD` request is made on a `:LISP-SOURCE` module. The first thing that the handler does is to initiate a `:COMPILE` request on each of the grains in the `:LISP-SOURCE` module, and then it models the fact that the `:BINARY` grains produced by compilation need to be loaded.

Handlers ensure that task graph paths exist. After a handler has been invoked on a grain once, additional invocations will have no effect. Therefore, task definers need only be concerned that the proper handlers are invoked at least once and do not need to worry about additional invocations.

```
(DEFINE-REQUEST-HANDLER (:COMPILE :LISP-SOURCE :PRE) (SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE :LISP-COMPILE) BINARY)))

(DEFINE-REQUEST-HANDLER (:LOAD :LISP-SOURCE :PRE) (SOURCE-NODE)
  (PROCESS-REQUEST :COMPILE SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE :LISP-COMPILE) BINARY
                        (BINARY :LISP-LOAD-BIN) IMAGE)))
```

Figure 5-1: Request Handler Definitions for Lisp

5.3 Reference Handlers

Reference handlers realize the implications references upon construction graphs. The construction implications of a reference depend upon the kind of reference, the request, and which part of the reference (right or left) the module participating in the request belongs to. Each handler is identified by a reference handler signature that includes five fields: the three fields from the reference signature, the request name, and a participation marker (either `:RIGHT` or `:LEFT`). Sample signatures are:

```
<<:CALLS :LISP-SOURCE :LISP-SOURCE> <:LOAD :LEFT>>
<<:CALLS :C-SOURCE :C-SOURCE> <:COMPILE :RIGHT>>
<<:MACRO-CALLS :LISP-SOURCE :LISP-SOURCE> <:COMPILE :LEFT>>
```

Not all references are relevant to every request made. For instance, the reference

```
(:CALLS LISP-SOURCE-1 LISP-SOURCE-2)
```

has no implications when a request is made to compile `LISP-SOURCE-1`. However, if the request is to load `LISP-SOURCE-1` for execution, then the reference implies that `LISP-SOURCE-2` needs to be loaded. It is also important to recognize that the direction of the reference matters. For example, the reference above has implications when `LISP-SOURCE-1` is loaded, but, it has none when `LISP-SOURCE-2` is loaded.

Defining Reference Handlers

Reference handlers are defined with DEFINE-REFERENCE-HANDLER. Calls have the form:

```
(DEFINE-REFERENCE-HANDLER ((REFERENCE LEFT-TYPE RIGHT-TYPE)
                           (REQUEST DIRECTION))
                          (ARGS)
  &BODY BODY)
```

REFERENCE The name of the reference being handled.

LEFT-TYPE The grain type of the left (first) module in the reference.

RIGHT-TYPE The grain type of the right (second) module in the reference.

REQUEST The name of the request being handled.

DIRECTION Either :LEFT or :RIGHT. This field identifies the module that the request being handled refers to.

ARGS The names of the variables passed to the handler, these will be bound when *BODY* is evaluated. There must be at least two elements in this list. The first *ARG* will be bound to the left grain of the reference. The second *ARG* will be bound to the right grain of the reference.

BODY The forms that constitute the handler. They are evaluated with the arguments passed to the handler bound to the variables named in *ARGS*.

Figure 5-2 contains reference handler definitions for Lisp compilation and loading. The first handler models the fact that the grain represented by CALLED-NODE needs to be loaded, and that the resulting :LISP-IMAGE grain plays the role DEFINITIONS in the compilation of the grain represented by CALLING-NODE. The second handler ensures that the grain represented by :CALLED-NODE is loaded. Note, while these handlers are sufficient to handle the common module interactions for Lisp systems, they are not sufficient to handle all of the ways that Lisp modules may interact. More handlers would need to be defined in order to properly handle all of the ways that Lisp modules can interact. The prototype implementation of BUILD does not include these additional handlers at this time.

BUILD guarantees that reference handlers are invoked after pre-reference request processing and therefore handler writers may safely assume that the effects of pre-reference request handlers will already be present in the graph. For example, the :MACRO-CALLS handler discussed above assumes that the compilation of CALLING-NODE has already been modeled.

5.4 A Task Description Definition Example

This section presents an example of a task description definition. The task defined is called :LIST-SOURCE-CODE and it will produce formatted source code listings for a :LISP-SOURCE module and any :LISP-SOURCE modules that it references. All of the defining forms for :LIST-SOURCE-CODE are in figure 5-3.

First, the :LIST-LISP-SOURCE process type is defined. Instances of this type have a single input role called SOURCE and a single output role called LISTING. The function LIST-LISP-FILE is called to produce the grain filling the output role from the grain filling the input role. The request handler for the task is very simple, it models the fact that the source grain to be listed will play the role SOURCE in a :LIST-LISP-SOURCE p-node and that a g-node should be attached to the LISTING role of that same

p-node. The two reference handlers specify that grains which are called by a grain being listed should themselves be listed.

:LIST-SOURCE-CODE shows the virtue of keeping system models separate from information about tasks: once its defining forms are evaluated, formatted listings may be obtained for any previously modeled Lisp system without altering any system models.

```
(DEFINE-REFERENCE-HANDLER ((:MACRO-CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:COMPILE :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (PROCESS-REQUEST :LOAD CALLED-NODE)
  (PUSH (ACCESS CALLED-NODE ((SOURCE :LISP-COMPILE) BINARY
                             (BINARY :LISP-LOAD-BIN) IMAGE))
        (ACCESS+ CALLING-NODE ((SOURCE :LISP-COMPILE) DEFINITIONS))))

(DEFINE-REFERENCE-HANDLER ((:CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:LOAD :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LOAD CALLED-NODE))
```

Figure 5-2: Reference Handler Definitions for Lisp

```
(DEFINE-PROCESS-TYPE :LIST-LISP-SOURCE
    ((SOURCE :LISP-SOURCE :SINGLE))
    ((LISTING :PRESS :SINGLE SOURCE))
    OUTPUT-STREAM
    (FORMAT OUTPUT-STREAM "~%LIST ~A"
            (PATHNAME-MINUS-VERSION SOURCE))
  (FORMAT OUTPUT-STREAM "~%LISTING ~A" SOURCE)
  (LIST-LISP-FILE SOURCE LISTING))

(DEFINE-REQUEST-HANDLER (:LIST-SOURCE-CODE :LISP-SOURCE :PRE)
                        (SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE :LIST-LISP-SOURCE) LISTING)))

(DEFINE-REFERENCE-HANDLER ((:MACRO-CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:LIST-SOURCE-CODE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LIST-SOURCE-CODE CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :LISP-SOURCE :LISP-SOURCE)
                           (:LIST-SOURCE-CODE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LIST-SOURCE-CODE CALLED-NODE))
```

Figure 5-3: Definition For :LIST-SOURCE-CODE

6. Reprise

This chapter highlights several aspects of BUILD that have been presented in this report. The first section summarizes how BUILD overcomes the difficulties associated with existing tools (see chapter 2). The second section discusses BUILD's construction framework and how it provides a base for describing new tasks within a static framework that conceals many low level details from the task definer. The final section proposes ways that BUILD could be extended to provide capabilities not found in existing tools.

6.1 BUILD Compared With Existing Tools

Phrased in terms of inter-module references. The BUILD system modeling mechanism allows users to describe systems in terms that are natural for them. BUILD system models are easier to understand and they provide more information than the construction directive lists used by existing tools.

User definable task descriptions. BUILD's task description mechanism is responsible for the fact that BUILD is not constrained to some embedded set of tasks. By separating system models and task descriptions, BUILD's knowledge about construction can be modified without requiring that system models be changed. However, if a new task is sensitive to a class of references previously ignored, then existing models will have to be updated.

Intermediate grains are not referenced. The only grains that are referred to in a system are the source grains that comprise modules. While intermediate grains are used in BUILD's task graphs, these grains never appear in system models.

All source grains must be referenced. All of the source grains that participate in a system either reference other grains in the system or are referenced by other grains in the system. Therefore, since BUILD models encode system referencing patterns, all of the source grains in a system must appear in any well formed BUILD model of that system.

6.2 BUILD's Construction Framework

BUILD provides procedures which guide the construction process. These procedures include hooks for the components of user supplied task descriptions. The set of fixed procedures take care of low level construction details that are common to all tasks and allow task definitions to contain just the details that are relevant to the particular task being defined.

The task graph representation and analysis algorithm provide a uniform way to describe and perform system maintenance tasks. New process types and grain types can easily be integrated into task graphs.

The ACCESS family of functions provide a general way for viewing and manipulating task graphs that isolates handler definitions from the low level mechanics of instantiating nodes, matching grain types between g-nodes and p-node ports, and actually linking nodes together.

The task graph derivation algorithm ensures that pre-reference request handlers are invoked before reference handlers and that reference handlers are invoked before post-reference request handlers. This algorithm is also responsible for translating module references into a series of handler invocations, one for each grain involved in a reference. Finally, the task graph derivation algorithm ensures that circular references (i.e., `(:CALLS A B) (:CALLS B A)`) do not cause infinite loops during reference handling.

BUILD's construction framework allows task definers to concentrate on the significant details of the task being defined (e.g., what process and grain types are used, what references are relevant and how should they be handled etc.) and isolates them from low level details (e.g., task graph analysis, node instantiation etc.).

6.3 Extensions to BUILD

BUILD provides a more graceful way of modeling systems than existing tools, yet it does not provide greater capabilities. This section proposes extensions to BUILD that would allow it to provide a set of facilities that other tools do not. The extensions are automatic derivation of system specifications from source code, support for patching and similar maintenance styles, and the incorporation of the nature of module change into the reconstruction algorithms.

Automatic Derivation of System Descriptions

The BUILD modeling mechanism provides a natural way to describe systems but it does not ensure that the descriptions are complete or correct. Designers are still required to generate system models by hand. A tool that could derive system models from source code would relieve designers of the chore of building system description files.

For simple languages, an analyzer could build a great deal of the model and locate areas that might present difficulties. For example, in most C systems all of the dependencies are caused by use of the `#INCLUDE` compiler directive and calls to externally defined symbols -- the reference assertions from these references could be synthesized automatically.

While there may be programming environments in which it is possible to mechanize the derivation of system models there are certainly languages for which such derivation would become arbitrarily complex. For example, Pitman develops an argument against automatic derivation of Lisp system models based on the complications caused by macros [Pitman 84].

Patching

There are many instances where a system maintainer may want to introduce changes into a system without making sure that the resulting system is consistent; for example, debugging experiments where small changes are introduced to examine some small part of the system. These changes may not be intended to become part of a released system, it may even be known that they will cause compilation of some other module to fail. Another instance where the ability to patch a system is important is when a quick fix is being attempted and it is important that the effects be seen quickly. This kind of change represents a tentative guess on the part of the maintainer. The introduction of such changes into systems must be supported by system management tools if such tools are going to help and not hinder maintainers.

The DEFSYSTEM patch facility provides some support for producing inconsistent systems. Unfortunately, the DEFSYSTEM patching facility makes no use of the dependency information that the rest of the tool uses. No analysis of the effect of a patch is available. Nothing guarantees that a patch will even be loaded correctly according to the dependency information that is available. For example, if a patch file includes a modified macro definition and two calls to it, the calls will not refer to the new version of the macro unless they are placed after the definition in the patch file by the user.

System management tools should make use of system models in order to support patching. Patching mechanisms should also supply information about the effect that a patch may have on the rest of the system. In BUILD, the analysis could be done by propagating the effects of a change through a task graph and then identifying those modules that were affected by the change but ignored by the patch.

More Precise Change Analysis

All of the tools mentioned in this paper (including BUILD) are sensitive to the fact that some change has occurred to a module in a system. However, no attention is paid to the nature of the change. By exploring the nature of a change it is possible to limit the amount of processing done when updating systems.

If source code is changed in a way that cannot alter its compilation, there is no reason for the source module to be recompiled. For example, compilation should not be done when source code has only been

reformatted or had commentary added to it. If a function is added to a module, but no existing modules are updated to contain calls to the new function, nothing should be done to the existing modules. Lint libraries are dependent upon the first pass of Lint, however, most changes to the first pass of Lint will not affect the libraries.

Change analysis can also provide important debugging information. For example, if a module interface is changed, but not all of the modules that contain references to that module are changed, there is a possibility that an error of omission has been made.

Unlike MAKE and BUILD, DEFSYSTEM can be extended to include more complicated predicates for deciding when changes are significant. There is nothing preventing a DEFSYSTEM system definition from using parsers and source code comparison programs in order to decide when transformations should take place. However, no enhanced predicates are supplied with DEFSYSTEM and none of the DEFSYSTEM descriptions encountered while preparing this paper included definitions of such specialized predicates.

Specialized predicates can only be useful if they require less processing to determine that a transformation can be avoided than applying the transformation in the first place. For instance, there is no point in using a predicate to determine that compilation of a module can be avoided if that predicate requires more processing than the compiler. BUILD can step around this issue by assuming that it is a single tool embedded in an integrated environment in which the tools that are used to modify modules can supply information to BUILD about the nature of changes.

BUILD could be extended to provide an interface for communicating information about changes to modules. The information passed to BUILD would include the name of the grain modified, the kind of modification made, and the name of the new (i.e., updated) grain. A new class of handlers called *change handlers* would be introduced to aid in the determination of *significant* changes by the construction algorithm.

For example, the change assertion:

```
(:ADDED-STRUCT DEFS)
```

would inform BUILD that DEFS has been changed by adding a new structure and therefore modules that rely on DEFS do not have to be re-compiled. The compilation of unaltered modules can be avoided since there is no way for them to refer to the new structure. The assertions:

```
(:ADDED-COMMENT DEFS)
(:RE-FORMATTED DEFS)
```

imply that no changes that can alter the compilation of DEFS have been made and therefore no re-compilation needs to be done.

The change handlers would contain listings of how types of changes alter the way in which grains play their roles. For instance, one handler would note that re-formatting a piece of source code does not change the way that it plays the role SOURCE in instances of :LISP-COMPILE.

References

[Ada 83]
Reference Manual For the Ada Programming Language.
United States Department of Defense, 1983.
Ansi/Mil-Std 1815 A

[Cooprider 79]
Cooprider.
The Representation of Families of Software Systems.
PhD thesis, Carnegie-Mellon University, April, 1979.

[Cristofor, et. al. 80]
Cristofor, Wendt, and Wonsiewicz.
Source Control + Tools = Stable Systems.
In *Proceedings of the Fourth Computer Software and Applications Conference*, pages pp. 527-532. IEEE, October, 1980.

[DeRemer and Kron 76]
DeRemer and Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2):80-86, June, 1976.

[Feldman 79]
Feldman.
Make - A Program for Maintaining Computer Programs.
Software - Practice and Experience 9(3):pp. 255-265, March, 1979.

[Horsley and Lynch 79]
Horsley and Lynch.
Pilot: A Software Engineering Case Study.
In *Proceedings of the 4th International Conference on Software Engineering*, pages 94-99. IEEE, September, 1979.

[Johnson 78a]
Johnson.
YACC - Yet Another Compiler Compiler.
Technical Report, Bell Laboratories, 1978.

[Johnson 78b]
Johnson.
Lint, a C Program Checker.
Technical Report, Bell Laboratories, July, 1978.

[Kernighan and Ritchie 78]
Kernighan and Ritchie.
The C Programming Language.
Bell Laboratories, 1978.

[Lewis 83]
Lewis.
Experience With A System For Controlling Software Versions In A Distributed Environment.
In *Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development*, pages 210-219. IEEE, November, 1983.

[Liskov 81]
Liskov et. al.
CLU Reference Manual.
1981.
Volume 114 of the Springer Verlag Lecture Notes in Computer Science

[Mitchell 79]
Mitchell, Maybury, Sweet.
Mesa Language Manual.
Fifth edition, XEROX PARC, 1979.

[Pitman 84]
Pitman.
The Description Of Large Systems.
Technical Report AI Memo 801, MIT Artificial Intelligence Laboratory, 1984.

[Rochkind 75]
Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering 1(4):pp 364-370, December, 1975.

[Schmidt 82]
Schmidt.
Controlling Large Software Development In a Distributed Environment.
PhD thesis, University of California Berkeley, December, 1982.
This thesis is available as XEROX PARC Technical Report CSL-82-7.

[Steele 84]
Guy Steele Jr.
Common LISP: The Language.
Digital Press, 1984.

[Tichy 80]
Tichy.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, January, 1980.

[Tichy 84]
Tichy.
RCS - A System for Version Control.
Technical Report CSD-TR-474, Purdue University, March, 1984.

[Weinreb and Moon 81]
Weinreb and Moon.
Lisp Machine Manual.
Fourth edition, Massachusetts Institute of Technology, 1981.

I. BUILD Definitions For C

The definitions used by BUILD to model a Lisp environment have been given in the body of this report as examples. This appendix contains the definitions used by BUILD to model a C programming environment. There are more kinds of commonly used grain types in UNIX environments than in Lisp environments, hence there are more definitions needed to model all of the ways that UNIX grains can refer to each other. Commentary has been added to highlight the definitions.

Grain Type Definitions

```
(DEFINE-GRAIN-TYPE :YACC-GRAMMAR :Y)
(DEFINE-GRAIN-TYPE :C-SOURCE :C)
(DEFINE-GRAIN-TYPE :C-OBJECT :O)
(DEFINE-GRAIN-TYPE :C-EXECUTE :EXE)
(DEFINE-GRAIN-TYPE :SHELL-SCRIPT :SCRIPT)
```

Process Type Definitions

It is assumed that the functions C-COMPILE, C-LOAD, and YACC are available.

```
(DEFINE-PROCESS-TYPE C-COMPILE
    ((SOURCE :C-SOURCE :SINGLE) (INCLUDES :C-SOURCE :MULTIPLE))
    ((OBJECT :C-OBJECT :SINGLE SOURCE))
    STREAM
    (FORMAT STREAM "~%COMPILE ~A" (PATHNAME-MINUS-VERSION SOURCE))
  (FORMAT STREAM "~%COMPILING ~A" SOURCE)
  (C-COMPILE SOURCE OBJECT))

(DEFINE-PROCESS-TYPE C-LOAD
    ((PRIMARY :C-OBJECT :SINGLE) (SECONDARY :C-OBJECT :MULTIPLE))
    ((IMAGE :C-EXECUTE :SINGLE PRIMARY))
    STREAM
    (FORMAT STREAM "~%LINK: ~A ~{~%  ~A~}"
            (PATHNAME-MINUS-VERSION PRIMARY)
            (MAPCAR #'PATHNAME-MINUS-VERSION SECONDARY))
  (FORMAT STREAM "~%LINKING: ~A ~{~%  ~A~}" PRIMARY SECONDARY)
  (C-LOAD PRIMARY SECONDARY IMAGE))

(DEFINE-PROCESS-TYPE YACC
    ((GRAMMAR :YACC-GRAMMAR :SINGLE))
    ((PARSER :C-SOURCE :SINGLE GRAMMAR))
    STREAM
    (FORMAT STREAM "~%YACC ~A" (PATHNAME-MINUS-VERSION GRAMMAR))
  (FORMAT STREAM "~%YACCING ~A" GRAMMAR)
  (YACC GRAMMAR PARSER))
```

Request and Reference Handlers

The request handler for C compilation models the fact that the source grain needs to be compiled. The only reference that can have an effect on C compilation is :INCLUDES. If GRAIN-1 includes GRAIN-2, then GRAIN-1 indirectly includes any grains that GRAIN-2 includes. The task :INCLUDE+ (described later) is responsible for gathering all of the grains included indirectly by a grain and attaching the corresponding g-nodes to the INCLUDES port of the :C-COMPILE p-node for the grain being compiled.

```
;;;
;;;         :COMPILE :C-SOURCE
;;;

(DEFINE-REQUEST-HANDLER (:COMPILE :C-SOURCE :PRE) (SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE C-COMPILE) OBJECT)))

(DEFINE-REFERENCE-HANDLER ((:INCLUDES :C-SOURCE :C-SOURCE) (:COMPILE :LEFT))
                          (INCLUDING-NODE INCLUDED-NODE)
  (LET ((COMPILE-PROCESS (ACCESS INCLUDING-NODE ((SOURCE C-COMPILE)))))
    (PUSH INCLUDED-NODE (ACCESS+ COMPILE-PROCESS (INCLUDES)))
    (PROCESS-REQUEST :INCLUDE+ INCLUDED-NODE COMPILE-PROCESS)))
```

If a :C-SOURCE grain calls another grain, then BUILD pessimistically assumes that it indirectly calls any grain called by the second grain. The task :LOAD+ gathers all of the grains called indirectly by a grain in order to ensure that the proper set of grains is linked together. The lack of a task like :LOAD+ in Lisp is due to the fact that in Lisp environments, grains are loaded incrementally instead of being explicitly linked together.

```
;;;
;;;         :LOAD :C-SOURCE
;;;

(DEFINE-REQUEST-HANDLER (:LOAD :C-SOURCE :PRE) (SOURCE-NODE)
  (PROCESS-REQUEST :COMPILE SOURCE-NODE)
  (ACCESS* SOURCE-NODE ((SOURCE C-COMPILE) OBJECT (PRIMARY C-LOAD) IMAGE)))

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-SOURCE :C-SOURCE) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS
          (ACCESS CALLING-NODE ((SOURCE C-COMPILE) OBJECT (PRIMARY C-LOAD)))))
    (PROCESS-REQUEST :COMPILE CALLED-NODE)
    (PUSH (ACCESS CALLED-NODE ((SOURCE C-COMPILE) OBJECT))
          (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-SOURCE :C-OBJECT) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS
          (ACCESS CALLING-NODE ((SOURCE C-COMPILE) OBJECT (PRIMARY C-LOAD)))))
    (PUSH CALLED-NODE (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))
```

Sometimes compiled objects are used as source grains (e.g. supplied libraries). These definitions encode the knowledge needed to handle the loading of :C-OBJECT grains.

```
;;;
;;;         :LOAD :C-OBJECT
;;;

(DEFINE-REQUEST-HANDLER (:LOAD :C-OBJECT :PRE) (OBJECT-NODE)
  (ACCESS* OBJECT-NODE ((PRIMARY C-LOAD) IMAGE)))
```

```
(DEFINE-REFERENCE-HANDLER ((:CALLS :C-OBJECT :C-SOURCE) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS (ACCESS CALLING-NODE ((PRIMARY C-LOAD)))))
    (PROCESS-REQUEST :COMPILE CALLED-NODE)
    (PUSH (ACCESS CALLED-NODE ((SOURCE C-COMPILE) OBJECT))
          (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-OBJECT :C-OBJECT) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS (ACCESS CALLING-NODE ((PRIMARY C-LOAD)))))
    (PUSH CALLED-NODE (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))
```

Here are the handlers for :INCLUDE+ and :LOAD+. There are no request handlers associated with these requests as all of the significant construction information that they imply arises from references. These handlers illustrate the use of more than two values being passed to reference handlers. The additional parameter for :INCLUDE+ is the :C-COMPILE p-node which models the compilation to be done. The additional parameter for :LOAD+ is the p-node which models the linking to be done.

```
;;;
;;;         :INCLUDE+ :C-SOURCE C-COMPILE
;;;

(DEFINE-REFERENCE-HANDLER ((:INCLUDES :C-SOURCE :C-SOURCE) (:INCLUDE+ :LEFT))
                          (IGNORE INCLUDED-NODE INCLUDING-PROCESS)
  (PUSH INCLUDED-NODE (ACCESS+ INCLUDING-PROCESS (INCLUDES)))
  (PROCESS-REQUEST :INCLUDE+ INCLUDED-NODE INCLUDING-PROCESS))

;;;
;;;         :LOAD+ :C-SOURCE C-LOAD
;;;

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-SOURCE :C-SOURCE) (:LOAD+ :LEFT))
                          (IGNORE CALLED-NODE LINKING-PROCESS)
  (PROCESS-REQUEST :COMPILE CALLED-NODE)
  (PUSH (ACCESS CALLED-NODE ((SOURCE C-COMPILE) OBJECT))
        (ACCESS+ LINKING-PROCESS (SECONDARY)))
  (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS))

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-SOURCE :C-OBJECT) (:LOAD+ :LEFT))
                          (IGNORE CALLED-NODE LINKING-PROCESS)
  (PUSH CALLED-NODE (ACCESS+ LINKING-PROCESS (SECONDARY)))
  (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS))

;;;
;;;         :LOAD+ :C-OBJECT C-LOAD
;;;

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-OBJECT :C-SOURCE) (:LOAD+ :LEFT))
                          (IGNORE CALLED-NODE LINKING-PROCESS)
  (PROCESS-REQUEST :COMPILE CALLED-NODE)
  (PUSH (ACCESS CALLED-NODE ((SOURCE C-COMPILE) OBJECT))
        (ACCESS+ LINKING-PROCESS (SECONDARY)))
  (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS))

(DEFINE-REFERENCE-HANDLER ((:CALLS :C-OBJECT :C-OBJECT) (:LOAD+ :LEFT))
                          (IGNORE CALLED-NODE LINKING-PROCESS)
  (PUSH CALLED-NODE (ACCESS+ LINKING-PROCESS (SECONDARY)))
  (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS))
```

Here are the definitions used to model YACC's interaction with C systems. The handlers capture the fact that YACC grammars may include and call other grains.

```
;;;
;;; :YACC :YACC-GRAMMAR
;;;

(DEFINE-REQUEST-HANDLER (:YACC :YACC-GRAMMAR :PRE) (GRAMMAR-NODE)
  (ACCESS* GRAMMAR-NODE ((GRAMMAR YACC) PARSER)))

;;;
;;;         :COMPILE :YACC-GRAMMAR
;;;

(DEFINE-REQUEST-HANDLER (:COMPILE :YACC-GRAMMAR :PRE) (GRAMMAR-NODE)
  (PROCESS-REQUEST :YACC GRAMMAR-NODE)
  (ACCESS* GRAMMAR-NODE ((GRAMMAR YACC) PARSER (SOURCE C-COMPILE) OBJECT)))

(DEFINE-REFERENCE-HANDLER ((:INCLUDES :YACC-GRAMMAR :C-SOURCE) (:COMPILE :LEFT))
                          (INCLUDING-NODE INCLUDED-NODE)
  (LET ((COMPILE-PROCESS
          (ACCESS INCLUDING-NODE ((GRAMMAR YACC) PARSER (SOURCE C-COMPILE)))))
    (PUSH INCLUDED-NODE (ACCESS+ COMPILE-PROCESS (INCLUDES)))
    (PROCESS-REQUEST :INCLUDE+ INCLUDED-NODE COMPILE-PROCESS)))

;;;
;;;         :LOAD :YACC-GRAMMAR
;;;

(DEFINE-REQUEST-HANDLER (:LOAD :YACC-GRAMMAR :PRE) (GRAMMAR-NODE)
  (PROCESS-REQUEST :COMPILE GRAMMAR-NODE)
  (ACCESS* GRAMMAR-NODE ((GRAMMAR YACC) PARSER
                         (SOURCE C-COMPILE) OBJECT
                         (PRIMARY C-LOAD) IMAGE)))

(DEFINE-REFERENCE-HANDLER ((:CALLS :YACC-GRAMMAR :C-SOURCE) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS (ACCESS CALLING-NODE ((GRAMMAR YACC) PARSER
                                               (SOURCE C-COMPILE) OBJECT
                                               (PRIMARY C-LOAD))))))
    (PROCESS-REQUEST :COMPILE CALLED-NODE)
    (PUSH (ACCESS CALLED-NODE ((SOURCE C-COMPILE) OBJECT))
          (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))

(DEFINE-REFERENCE-HANDLER ((:CALLS :YACC-GRAMMAR :C-OBJECT) (:LOAD :LEFT))
                          (CALLING-NODE CALLED-NODE)
  (LET ((LINKING-PROCESS (ACCESS CALLING-NODE ((GRAMMAR YACC) PARSER
                                               (SOURCE C-COMPILE) OBJECT
                                               (PRIMARY C-LOAD))))))
    (PUSH CALLED-NODE (ACCESS+ LINKING-PROCESS (SECONDARY)))
    (PROCESS-REQUEST :LOAD+ CALLED-NODE LINKING-PROCESS)))
```

Here are the definitions used to handle :SHELL-SCRIPT grains. A request to compile or load a shell script is interpreted to mean that all of the modules called by the script should be compiled or loaded.

```
;;;
;;;         :COMPILE :SHELL-SCRIPT
;;;

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :C-SOURCE) (:COMPILE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :COMPILE CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :C-OBJECT) (:COMPILE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :COMPILE CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :YACC-GRAMMAR) (:COMPILE :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :COMPILE CALLED-NODE))

;;;
;;;         :LOAD :SHELL-SCRIPT
;;;

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :C-SOURCE) (:LOAD :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LOAD CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :C-OBJECT) (:LOAD :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LOAD CALLED-NODE))

(DEFINE-REFERENCE-HANDLER ((:CALLS :SHELL-SCRIPT :YACC-GRAMMAR) (:LOAD :LEFT))
                          (IGNORE CALLED-NODE)
  (PROCESS-REQUEST :LOAD CALLED-NODE))
```